CHARACTERIZATION AND DESIGN OF HIGH LEVEL VHDL I/Q

FREQUENCY DOWNCONVERTER VIA SPECIAL SAMPLING SCHEME

THESIS

JESSE P SOMANN, SECOND LIEUTENANT, USAF

AFIT/GE/ENG/06-53

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/06-53

CHARACTERIZATION AND DESIGN OF HIGH LEVEL VHDL I/Q FREQUENCY

DOWNCONVERTER VIA SPECIAL SAMPLING SCHEME

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering
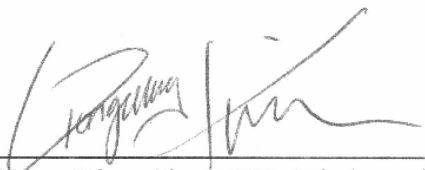
Jesse P. Somann

Second Lieutenant, USAF

March 2006

AFIT/GE/ENG/06-53

CHARACTERIZATION AND DESIGN OF HIGH LEVEL VHDL I/Q FREQUENCY

DOWNCONVERTER VIA SPECIAL SAMPLING SCHEME

Jesse P Somann
Second Lieutenant, USAF

Approved:

_____   _____
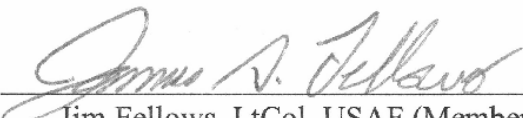Yong Kim, Civ, AFIT (Chairman)   2 MAR '06
                                                         date

_____   _____
Gregory Creech, Civ, AFRL (Member)   2 MARCH 06
                                                         date

_____   _____
Jim Fellows, LtCol, USAF (Member)   2 MAR 06
                                                         date
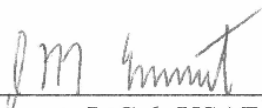
_____   _____
Marty Emmert, LtCol, USAF-Res (Member)   2 March 06
                                                         date

Acknowledgements

I thank Dr. Yong Kim, my thesis advisor, for all his effort and guidance in support of this research. The constant encouragement was very appreciated. I would also like to send my sincere appreciation to Dr. Marty Emmert for the large amount of time and effort he spent sharing his technical knowledge and guiding me through finer points of this research. A thank you also to my sponsor, Dr. Greg Creech of the Air Force Research Laboratory for the continued support in this endeavor.

I would also like to thank my classmates who supported and helped me throughout the thesis class load, and research and writing process: Lt Chris Wood, Lt Mark Martin, Maj Brad Christiansen, and many others.

Jesse P. Somann

Table of Contents

Table of Figures

Table of Tables

List of Acronyms

AFIT/GE/ENG/06-53

Abstract

This study explores the characterization and implementation of a Special Sampling Scheme (SSS) for In-Phase and Quad-Phase (I/Q) downconversion utilizing top-level, portable design strategies. The SSS is an under-developed signal sampling methodology that can be used with military and industry receiver systems, specifically, United States Air Force (USAF) video receiver systems. The SSS processes a digital input signal-stream sampled at a specified sampling frequency, and downconverts it into In-Phase (I) and Quad-Phase (Q) output signal-streams. Using the theory and application of the SSS, there are three main objectives that will be accomplished: characterization of the effects of input, output, and filter coefficient parameters on the I/Q imbalances using the SSS; development and verification of abstract, top-level VHDL code of the I/Q SSS for hardware implementation; and finally, development, verification, and analysis of variation between synthesizable pipelined and sequential VHDL implementations of the SSS for Field Programmable Gate Arrays (FPGA) and Application Specific Integrated Circuits (ASIC).

CHARACTERIZATION AND DESIGN OF HIGH LEVEL VHDL I/Q FREQUENCY DOWNCONVERTER VIA SPECIAL SAMPLING SCHEME

I. Introduction

1.1 General Issue

Electronic Warfare systems have become a major research issue for the Department of Defense (DoD), and military research facilities such as the Air Force Research Laboratory (AFRL). Billions of dollars are put into developing newer and better systems of radar communication and processing. As we continue to advance in technology, the speed and capabilities of digital communications continue to increase. Through material advancements and with the help of some specialized methods of sampling, digital receivers are beginning to compete with analog devices in quality of signal reception.

Application Specific Integrated Circuits (ASICs) are ideal for their low production costs for mass produced circuits (after high initial design costs) when high-speed acceleration of a task is needed. However, ASICs force the circuit designer to make critical decisions very early in the product life-cycle, at initial design time. One may obtain very high speed by dedicating specialized logic elements, but losing the flexibility to adapt the circuit operation later. Or, one may choose instead to design-in some degree of generality for later adaptability by incorporating a specialized "microcoded" instruction set or "reconfigurable" parameters. While not as flexible as a

general-purpose CPU, such an instruction set preserves (to a degree based on design tradeoffs) the relatively higher speed of ASICs.

This adaptability comes at a cost, however. An ASIC circuit that is customized to a specific task, such as signal processing or communication hardware, can be designed to operate in a single cycle of the system clock by use of a pipeline design, and is thus very fast relative to conventional microprocessors in which many cycles are required to implement the same algorithm. When the generality of microcode is added to an ASIC the designer is forced to 1) accept that some algorithms will need multiple clock cycles (i.e., multiple instructions) to execute, thus slowing the microprocessor down, 2) use more of the available chip area than before, reducing the speed otherwise obtained via parallelism, and 3) predict at the start of product life-cycle the range of instructions needed to implement all anticipated improvements in signal processing or communication over the life of the product.

The more general the design, the more intrinsic ASIC speed is lost to the design. Also, ASICs force the designer to select a rigid datapath bit-width intrinsic to the design. If the datapath is too narrow, higher precision data must be double or multiple-clocked through the device, slowing the data down. If the full datapath width is not needed for a particular algorithm, space on the chip goes wasted: again, effectively slowing the system down if compared to the FPGA approach, which fully utilizes each chip's logic resources and in which path width is set dynamically.

Consequently, traditional ASICs are ideal for applications that are well-defined and not expected to change during the field operation of the product. The recent history of signal processing and communication, however, is characterized by rapid improvement

in the state-of-the-art, creating significant risk of early obsolescence for any ASIC based accelerator, even if microcoded.

In addition, it generally takes a long period of time to design a circuit for a specific radar application through conventional layout design for an ASIC. Significant financial and labor costs are often spent in order to re-design, or make multiple circuit designs that perform virtually identical functions of the original, but with slightly different parameters. In general, to utilize a new technology, or to change input or output parameters in a circuit, a layout must be done from scratch, often limiting the use of existing components. These issues cause long turn-around times and in many cases, wasted resources.

The purpose of this study is to provide a highly portable method of designing and generating usable signal processing circuits for faster turn around time and highly reduced cost. Specifically, the design method:

    1: Will have a shorter turn-around time from design to fabrication.

    2: Will be portable between fabrication technologies.

    3: Can be targeted for use with FPGA or ASIC devices.

1.2 Specific Issue

Frequency is a large liability in digital communication applications. There is always a frequency range that should be monitored, but in order to cover all frequency values, a signal may need to be split into parallel channels, a process which is called channelization. Frequency conversion can then be made to match the frequency of interest to that of the Intermediate Frequency (IF) receiver. Due to effects of second

harmonics that can occur, an IF frequency bandwidth is usually kept within an octave (meaning that the high end is twice the frequency of the low end).

One method for frequency conversion in digital communications is to split the input signal into two channels (in-phase and quadrature) that are 90-degrees out of phase from each other. This down conversion can simply be referred to as I and Q channelization. A receiver that uses such a system is referred to as a superheterodyne receiver. A superheterodyne receiver takes the incoming signal, mixes it with a local oscillator to create the I and Q channels which are at a constant IF, and then will run those signals through a faster demodulator to resurrect the original signal [1].

Advantages to using such a method in digital receivers include: the input signal bandwidth can be doubled, amplitude measures of the signal are maintained, and the channels can be used to determine instantaneous frequency with either the I or the Q [2].

One specific use for I/Q demodulation is for digital receiver systems. In 2000-2004, the AFRL commissioned Raytheon to help develop an advanced digital receiver and to demonstrate it's feasibility for high-speed digital processing. The receiver was to be in use in airborne radar and was to be extended to ground unit use as an X-band Radar (XBR) system. Referred to as the Modular Digital RF System (MoDRFS), the system included a receiver system with analog-to-digital conversion and I/Q preprocessor [3]. Raytheon demonstrated the feasibility of such a system with an FPGA implementation that utilized an unknown algorithm.

Our concern is with the I/Q data conversion module (preprocessor) of such a receiver. The purpose of the preprocessor is to decimate incoming digital data (at an IF), form it into I and Q data streams, and buffer and format the streams for use by the radar

16

processing modules. In the MoDRFS, the video processor unit itself is the heart of the

receiver system as it receives 16-bit data steams from the delta/sigma ($\Delta\Sigma$) Analog-to-

Digital Converter (ADC), delays those streams, filters the streams in two stages

(decimation filter and I and Q Programmable Video Filters (I/Q PVF)), and outputs the

resulting I/Q data. This type of application lends itself towards I/Q data processing [4].

I/Q formulation has many benefits in both military and industry applications.

First of all, the formulation can allow for sampling of a range of frequencies without a

need for complete redesign of an ASIC or FPGA. Only the sampling frequency of the

local oscillation needs to be changed.

In order for this process to be more valuable, design options are needed that do

not require significant time and effort to implement. A top-level design with parameters

that can be easily adjusted would prevent the need to create layouts from scratch,

therefore reducing time and costs. Also, a flow of design options for optimizing for size

versus speed would allow more flexibility in physical synthesis optimizations. More

options for synthesis will also result in a reduction of time and money spent by military

and industry on projects.

1.3 Problem Statement & Scope of Study

The problem to be solved is this: the characterization and implementation of a

SSS for I/Q downconversion utilizing top-level, portable design strategies. To keep the

overall design of a digital pre-processor module applicable, it must be able to be

implemented with any common industrial synthesis software package (Synopsys,

Cadence, Mentor Graphics, etc.). This need is due to the often changing contract

situations that occur within research and development labs. If AFRL were to switch to a new company's software, or uses a different software set from the same company, the design needs to be portable to work with those changes. VHDL is a basic language used by all such companies and therefore works to accomplish this requirement.

Such a design must also be done in a common and steady language, and in a basic high-level implementation due to ever changing levels of technology. For example, in the past, the highest (smallest) technology sizes were at the 0.5 µm and 0.25 µm ranges. Currently, sizes of 0.18 µm, 0.09 µm, and even as small as 0.065 µm are starting to be used. With these new sizes of technology come new parameter limitations due to size, power, speed, and interference. New sets of design rules now need to be applied to the same designs. Therefore an already laid out design cannot just be reduced by a 1-to-1 factor, but must be re-synthesized all together. With a high-level behavioral, or optimized structural VHDL design, each module in a processor circuit can be re-developed and synthesized in any major software package with limited effort and time.

For AFRL purposes, the I/Q channel demodulator must also be flexible in scope. They want to be able to adjust input, output, and internal filtering parameters, without significantly altering the quality of the I/Q data outputs. In order to accomplish this, we need to adequately characterize and parameterize the data so as to accommodate changes in parameters.

While there are many different methods of I/Q data sampling, most developed methods involve analog mixers. Two approaches to I/Q data via digital circuitry exist: the Hilbert transform, and a SSS. In the past, digital methods have not been reasonable to use in radar applications due to limitations in the speed of digital circuitry. Recently

18

however, with the advancement of technologies, digital capabilities are starting to compete with analog capabilities.

To create I/Q channels, we will be using a SSS that mixes the incoming signal and runs the mixed samples through lowpass filters to filter out the high frequencies produced by mixing. The SSS will be described in depth as we continue in the thesis. By analyzing Matlab simulations of sampled data using this SSS, we can fully characterize the effect of changing data parameters. The main concern of the changing parameters is that of I/Q data imbalance in the output, including differences in amplitude between the signals, and phase differences varying from the ideal 90 degrees between the I and Q channels.

Our implementation will be based on a strictly hardware basis. Hardware does not usually use complex mathematical types such as floating point values because of size and speed requirements. Instead we will concentrate on binary fixed-point math. One implication of this is that we must use only std_logic and std_logic_ vector input/outputs for the VHDL I/Q data formation. Because of this, bit lengths and widths of input/output signals and filter coefficients become very important.

1.4 Research Objectives

With the ever-evolving technology levels of today, we have a great need for abstract, easily portable design module libraries. These libraries must be well defined and verified. Thus, this study has three main objectives:

- Characterization of the effects of input/output and filter coefficient parameters using the SSS

- Top-level VHDL code of I/Q SSS for hardware implementation

- Variation and verification of synthesis through industry design flow for ASICs and FPGAs.

The characterization will be performed using Matlab functions and plots, and will later be verified by VHDL simulation and hardware FPGA testing. The key here will be to see how altering the input bits, coefficient bits, filter lengths, and even the output bits will effect the I/Q data. With the Matlab functions, we will use a base case model, and from there will alter the parameters and record the effects on I/Q amplitude differences and phase variations. The end goal is to create a set of trends and figures that will allow a designer to easily select design parameters that will meet their specific I/Q imbalance limitations.

Once characterized, the Matlab code will be converted into a high-level VHDL code using 'real' type variables (which utilizes the IEEE standard 64-bit floating point number scheme). We will use this implementation to verify the exact process that was characterized in Matlab using a VHDL simulation software such as Symphony and ModelSim. Once verified, we will continue with digital implementation by creating a new set of modules that use only 'std_logic' and 'std_logic_vector' types. These modules will be fully synthesizable, easily adjustable (for bit parameters and filter lengths) and will be implementable by an ASIC or FPGA. Using this code, we again verify it's operation with Matlab. Finally, we will implement a set of Perl scripts that can

be used to easily convert the desired parameters developed using the a Matlab process, into synthesizable VHDL modules.

The final step will be to put the VHDL code through a synthesis design flow. We will attempt to verify an optimized design by comparing three separate implementations, each tested with various sets of parameters. The first will use a pipelined implementation that should optimize speed. This should in effect, have the largest size, fastest data conversion speed, and most power consumption. Next we will use a design to optimize the size of the circuit by utilizing a sequential implementation. A sequential methodology should have the smallest size, but a much larger data conversion speed. Finally, we will test a top-level, abstract implementation that will allow the industry synthesis software to use its own method to optimize the design.

1.5 Chapter Summary

This research project attempts to take an already-started AFRL effort, and further it's use by making the project into a flexible and modular design. This design will allow a continued and varied use of a very under-developed sampling scheme towards a useful implementation. This project will continue to improve the optimization and efficiency of AFRL research by developing a new modular signal-processing component. The study will also find cost saving alternatives to current design flow methodologies, allowing for a possible reduction in license costs.

1.6 References

[1]   Introduction to Naval Weapons Engineering: Tutorial. "Superheterodyne
        Receivers." 04 November 2005 http://www.fas.org/man/dod-
        101/navy/docs/es310/superhet.htm .

[2]   Tsui, James. *Digital Techniques for Wideband Receivers.* Artech House, 2001.

[3]   Hong, T.K. and Bui, L.T. "Video Processor 2 (VP2) IC Functional
        Requirements." Technical Paper (Proprietary), Raytheon Systems Company.
        1999.

[4]   "Modular Digital Radio Frequency System (MoDRFS)." Final Report, Raytheon
        Systems Company. Government Contract: F33615-98-2-1330. 19 May, 2004.

II. Literature Review

2.1 Chapter Overview

This chapter provides an overview of the research involved in understanding the original AFRL video processor design plans. Information will also include research involved in implementing and analyzing the Special Sampling Scheme (SSS) including sampling and filtering, as well as that involved in creating a workable VHDL code and synthesizing that code into an FPGA or ASIC implementation.

2.2 The Video Processing Unit

The MoDRFS implemented by Raytheon for AFRL gives motivation for the development of basic I/Q sampling principles. It consists of elements including a high power active array antenna, a multi-channel receiver system, and other control units. The signals picked up by the antenna are transferred to the receiver unit as an X-band Radio Frequency (RF). The receiver unit consists of two components: the receiver and the preprocessor. A block diagram of the receiver unit is shown in Figure 2.1. The receiver module takes the RF signal, translates it into an IF signal, and runs that signal through an ADC. This digital signal is finally transferred to the preprocessor module where it is decimated and formed into I and Q data channels [1].
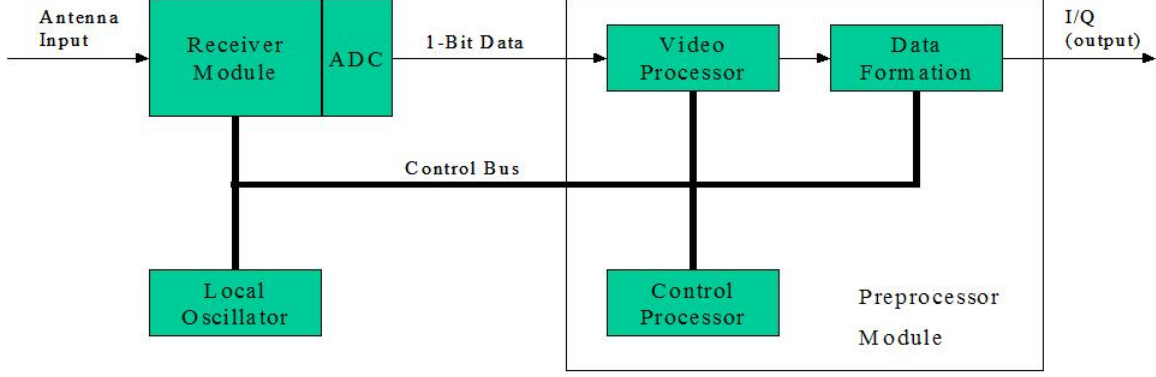
Figure 2.1: Block Diagram representation of a digital receiver unit like that in the MoDRFS.

2.3 I/Q Data Channels and The Special Sampling Scheme

In this section we will explore the basics behind the SSS method of digital

frequency downconversion that will be used to implement our circuit codes. To reiterate,

one method for frequency conversion in digital communications is to split the input

signal into two channels and can be referred to as I and Q channelization.

For example, if we start with a sinusoidal signal with amplitude A and frequency

f_i:

$$S(t) = A\sin(2\pi f_i t) \tag{2.1}$$

We can then generate our I and Q channels by using a local oscillator (sampling)

frequency f_o and filtering out the high frequency component:

$$I(t) = A\cos\left[2\pi(f_i - f_o)t\right] \tag{2.2}$$

$$Q(t) = A\sin\left[2\pi(f_i - f_o)t\right] \tag{2.3}$$

25

If these two signals are added, a complex data signal can be formed (I as real, Q imaginary, or vice versa).

Advantages to using such a method in digital receivers includes: the input signal bandwidth can be doubled, amplitude measures of the signal are maintained, and the channels can be used to determine instantaneous frequency with either the I or the Q [2].

One very common downfall to I and Q data are imbalances between the two. More specifically, the two channels will not always have the same amplitudes and will not always be perfectly $90°$ apart. Such an occurrence can negatively affect the dynamic range of the receiver processing the signal. Imbalances are more common in analog processing than they are in digital approaches. This is because in the digital techniques for I/Q generation, one signal (I or Q) is taken with a single-channel downconverter, and is then manipulated to create the out-of-phase channel. There are also digital correction methods of I/Q channels that are presented in detail in Tsui's book, page 276, and his references [2], but they involve a number of complex formulas and equations. Such a method can work for the back end of a system, but for hardware, it will be better to look into avoiding such imbalances from the start as we will do in Chapter 3. A preferred method will be to measure how changing bit-widths and binary math techniques in the processing hardware, may affect digitally generated I and Q data and their balance.

The real downfall to using a digital method is that slow processing speeds can limit the bandwidth that can be covered to a minimum. For military purposes (electronic warfare, monitoring, etc.) these bandwidths may be too low.

There are two methods to creating digital I/Q channels. One can either use a Hilbert transform [2], or a SSS. The later is the approach this study will use.

In a SSS, let us assume an input signal

$$V_i = A\sin\left(2\pi f_i t + \theta\right) \tag{2.4}$$

With amplitude A, initial phase θ, and known input frequency f_i. A sampling period of t_s is used, starting at $t=0$ and sampling at twice the Nyquist rate ($2*f_i$) or $4*f_i$. Therefore:

$$t_s = 1/\left(4*f_i\right) \tag{2.5}$$

The resulting signal is:

$$V_i = A\sin\theta, A\cos\theta, -A\sin\theta, -A\cos\theta, A\sin\theta...etc \tag{2.6}$$

Here we see that odd and even terms are $90°$ out of phase and so we can say that odd terms are in the I channel and even terms are in the Q channel:

$$\begin{aligned} I &= A\sin\theta, -A\sin\theta, A\sin\theta... \\ Q &= A\cos\theta, -A\cos\theta, A\cos\theta... \end{aligned} \tag{2.7}$$

While this is very intuitive and simplistic, it only works if the designer knows the input signal frequency, which in military applications is not always the case. Therefore, we must use a method that will work while knowing only our sampling frequency.

For a signal with an unknown frequency, V_{if}, let us look at the outputs when we run the signal through a local oscillator with two out-of-phase mixers:

$$V_{if}1 = \sin\left(2\pi f_i t + \theta\right)\cos\left(2\pi f_o t\right)$$
$$= \frac{1}{2}\left\{\sin\left[2\pi\left(f_i - f_o\right)t + \theta\right] + \sin\left[2\pi\left(f_i + f_o\right)t + \theta\right]\right\} \tag{2.8}$$
$$V_{if}2 = \sin\left(2\pi f_i t + \theta\right)\sin\left(2\pi f_o t\right)$$
$$= \frac{1}{2}\left\{\cos\left[2\pi\left(f_i - f_o\right)t + \theta\right] + \cos\left[2\pi\left(f_i + f_o\right)t + \theta\right]\right\} \tag{2.9}$$

in which the oscillator frequency is known as f_o, the input frequency (f_i) is not known, and the high-frequency component will be filtered out with a low pass filter. Then if we start at zero phase and sample the output at $t = 1/4 f_o$:

$$V_{if}1 = \sin\left(\theta\right), 0, -\sin\left(2\pi f_i t_2 + \theta\right), 0, \sin\left(2\pi f_i t_4 + \theta\right), 0... \tag{2.10}$$

$$V_{if}2 = 0, \sin\left(2\pi f_i t_1 + \theta\right), 0, -\sin\left(2\pi f_i t_3 + \theta\right), 0... \tag{2.11}$$

Combined into one data channel in time order:

$$V_{if} = \sin\left(\theta\right), \sin\left(2\pi f_i t_1 + \theta\right), -\sin\left(2\pi f_i t_2 + \theta\right), -\sin\left(2\pi f_i t_3 + \theta\right)...$$
$$= \sin\left(2\pi f_i t + \theta\right) \tag{2.12}$$

which is where we started from in (2.4).

Now, say we use the Nyquist sampling theorem on a sampling frequency f_s, which tells us that the input bandwidth then is $f_s/2$ at max. Because of this, our local oscillator frequency must be $f_o = f_s/4$ to generate Equation (2.10) and Equation (2.11). Using this however will produce a signal equivalent to that of Equation (2.8) and Equation (2.9). Again, we need to filter out the high frequency term of these equations and thus will implement a low pass filter with cutoff frequency of $f_s/2$, matching the input frequency limitation.

For demonstration purposes, let us say we have a signal coming in:

$$X(t) = x(1), x(2), x(3), x(4), x(5), x(6), x(7), x(8), x(9)... \qquad (2.13)$$

which divided into 2 groups with zeros added and made out-of-phase makes:

$$I(t) = x(1), 0, -x(3), 0, x(5), 0, -x(7)...$$
$$Q(t) = 0, x(2), 0, -x(4), 0, x(6), 0, -x(8)... \qquad (2.14)$$

In which both must be run through a low pass filter with coefficients (impulse response):

$$h(t) = h(1), h(2), h(3), h(4), h(5), h(6), h(7), h(8), h(9)... \qquad (2.15)$$

that will result in convolution outputs YI and YQ:

$$YI(t) = I(t) \otimes h(t)$$
$$YQ(t) = Q(t) \otimes h(t) \qquad (2.16)$$

where a convolution is represented with a $\otimes$ symbol. This will give the results we want. However, in hardware, it is better not to waste effort by calculating the zero terms that result from the sinusoidal pattern in the data, but rather to manipulate the input data in a different way.

With the above approach, the filter must do its job at the same rate as the input data is being sampled. To reduce the operation speed of the filters, we can eliminate the zero terms, and double the number of filters. This will cut the required filter operation speed in half.

The reason we can accomplish this is that by using the definition of convolution we find that the odd ordered YI (1), YI (3)… YQ(1), YQ(3)… use only odd terms of $h(t)$, and even ordered YI (2), YI (4)… YQ(2), YQ(4)… use only even terms of $h(t)$ [2]. This is due to the zeros in the original strings. Conveniently however, this sets us up to make a common sense adjustment to the input data sampling.

The signal $X(t)$ coming in as above, can now be sampled as:

$$\begin{aligned} X1(t) &= x(1), x(5), x(9)... \\ X2(t) &= x(2), x(6), x(10)... \\ X3(t) &= x(3), x(7), x(11)... \\ X4(t) &= x(4), x(8), x(12)... \end{aligned}$$

(2.17)

$X3$ and $X4$ can be inversed, and the signals $X1$ and $X3$, and $X2$ and $X4$ reformed into:

$$\begin{aligned} X_r(t) &= x(1), -x(3), x(5), -x(7)... \\ X_i(t) &= x(2), -x(4), x(6), -x(8)... \end{aligned}$$

(2.18)

as in Equation (2.14) but without the zeros. Next a filter design must be made and the filter coefficients calculated and separated into:

$$\begin{aligned} h_o(t) &= h(1), h(3), h(5), h(7)... \\ h_e(t) &= h(2), h(4), h(6), h(8)... \end{aligned}$$

(2.19)

Using these with Equation (2.18), four filters will be created to do the following operations:

$$Y_{ro}(t) = X_r(t) \otimes h_o(t) = Y_{ro}(1), Y_{ro}(2), Y_{ro}(3)...$$
$$Y_{re}(t) = X_r(t) \otimes h_e(t) = Y_{re}(1), Y_{re}(2), Y_{re}(3)...$$
$$Y_{io}(t) = X_i(t) \otimes h_o(t) = Y_{io}(1), Y_{io}(2), Y_{io}(3)...$$
$$Y_{ie}(t) = X_i(t) \otimes h_e(t) = Y_{ie}(1), Y_{ie}(2), Y_{ie}(3)...$$

$$(2.20)$$

Again we can manipulate the data and combine the filter outputs Y_{ro} and Y_{re}, and Y_{io} and Y_{ie}:

$$Y_r(t) = Y_{ro}(1), Y_{re}(1), Y_{ro}(2), Y_{re}(2), Y_{ro}(3), Y_{re}(3)...$$
$$= Y_r(1), Y_r(2), Y_r(3), Y_r(4), Y_r(5), Y_r(6)...$$
$$Y_i(t) = Y_{io}(1), Y_{ie}(1), Y_{io}(2), Y_{ie}(2), Y_{io}(3), Y_{ie}(3)...$$
$$= Y_i(1), Y_i(2), Y_i(3), Y_i(4), Y_i(5), Y_i(6)...$$

$$(2.21)$$

This is very close to what we are looking for from Equation (2.14), but $YQ(1)$ as has been figured in the above method, must be zero. The way Equation (2.21) is implemented, it is not. Therefore we must add the zero to the first term of Y_i(t), and shift the rest of the values to make the final signals:

$$Y_I(t) = Y_r(t) = Y_r(1), Y_r(2), Y_r(3)...$$
$$= Y_I(1), Y_I(2), Y_I(3), Y_I(4)...$$
$$Y_Q(t) = 0, Y_i(t) = 0, Y_i(1), Y_i(2), Y_i(3)...$$
$$= Y_Q(1), Y_Q(2), Y_Q(3), Y_Q(4)...$$

$$(2.22)$$

2.4 Digital Filtering

The SSS cannot be used unless a digital filter module is included in it's architecture. Therefore we must also understand the basic principles of digital filtering of

incoming data streams. Once we know how the filtering process works, we will be able to implement an effective filter design module into the SSS circuit design.

A signal can be described as a combination of sine or cosine functions at specific frequencies. By sampling the magnitude of the signal coming in at a certain interval of time, one can create what is called a digital signal, or a set of such magnitudes set on a time axis, that accurately represents the activity of the analog signal.

Certain frequencies of the signal can then be filtered out digitally by applying transfer functions (mathematical formulas) to the data points. An example of this is represented in Figure 2.2. A simple form of digital filter is referred to as a moving averager: see Figure 2.3. This type of design simply takes the present point value, and averages its amplitude with a set number of previous point values of the digital signal. The most basic averager can be thought of as a 'rectangular window' in which all present and past digital values are weighted the same in the average, or rather, are multiplied by the same coefficient. 'Rectangular' comes from the appearance of a rectangle in the time domain representation.

Such a filter is defined as an FIR or Finite Impulse Response filter. The formula for the response of a length N FIR is expressed as:

$$H(e^{(j\theta)}) = \sum_{k=0}^{N-1} \omega(k)e^{(-jk\theta)} \qquad (2.23)$$

where θ is phase, k is a certain sample of the signal, and ω represents the window that is being used. When filtering with the windowing method, the term $\omega(k)$ can be expressed depending on the shape of the window that is chosen for use. Such effects as

Gibbs phenomenon and weakened transition bands may occur, but should be able to be

dealt with by using an appropriate window [3].
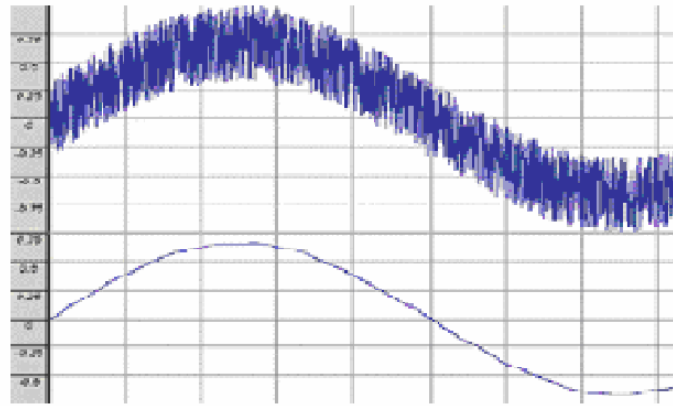
Figure 2.2: Digital filters remove unwanted frequencies such as noise from the upper
trace to produce the filtered lower trace.
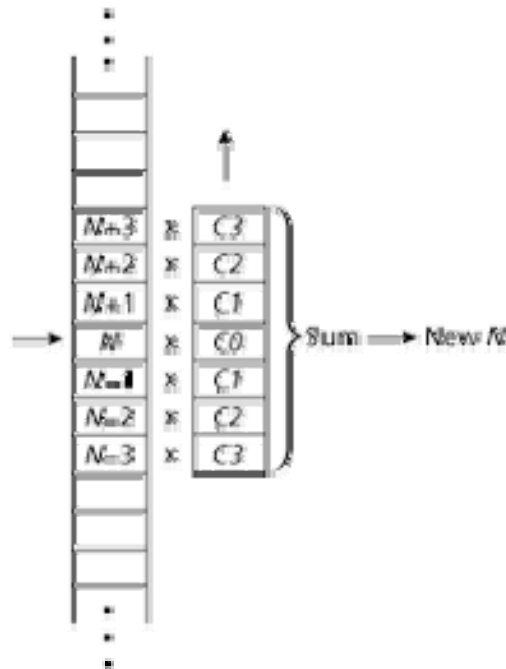
Figure 2.3: A moving average filter takes samples, sums them, and divides them by the
number of samples. This is a seven-point moving average.

A rectangular window is considered an ideal low pass filter, and as such, the window has a formula of:

$$\omega(k) = 1, \qquad 0 \le k \le N$$
$$= 0, \qquad \text{otherwise}$$

(2.24)

Such a filter will always have ripple, or artifact of abrupt window edges in transform, at the transition band edges due to the sudden drop from unity to zero in the time domain series. A smoother frequency response is created however, if we smooth out this abrupt drop in the time domain.

Smoothing out the window edges however, also has the side effects of Figure 2.4 that in essence say: if we adjust the filter to lower side lobe amplitude, we increase main lobe width, and if we decrease main lobe width we will also be increasing side lobe amplitude.

There have been a number of equations that can be used to create smoother windows for filtering with [3]. These include:

Bartlett (triangular):

$$\omega(k) = 2k/N, \qquad\qquad\qquad 0 \le k \le N/2$$
$$= 2 - 2k/N, \qquad\qquad\qquad N/2 \le k \le N$$
$$= 0, \qquad\qquad\qquad \text{otherwise}$$

(2.25)

Hann:

$$\omega(k) = 0.5 - 0.5\cos(2\pi k/N), \qquad\qquad 0 \le k \le N$$
$$= 0, \qquad\qquad\qquad \text{otherwise}$$

(2.26)

Hamming:

$$\omega(k) = 0.54 - 0.46\cos(2\pi k / N), \qquad 0 \le k \le N$$
$$= 0, \qquad \text{otherwise}$$

$$(2.27)$$

Blackman:

$$\omega(k) = 0.42 - 0.5\cos(2\pi k / N) + 0.08\cos(4\pi k / N), \qquad 0 \le k \le N$$
$$= 0, \qquad \text{otherwise}$$

$$(2.28)$$
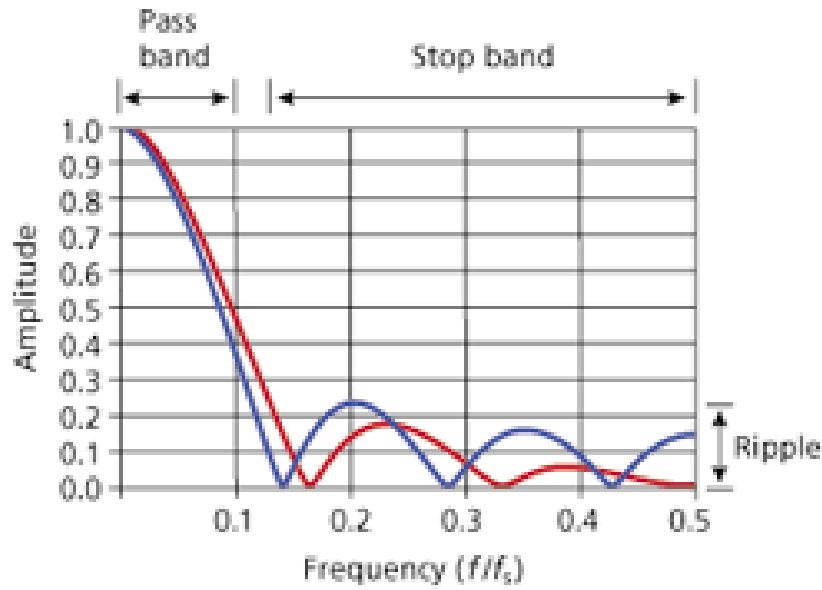
Figure 2.4: A seven-point averager that applies equal weights to each coefficient (blue), and varying weights to reduce ripple at the expense of a wider pass band (red).

The parameters of the different window's effects on the main lobe width and side lobe amplitudes can be seen in Table 2.1.

Table 2.1: Filter window type parameters [6].

WINDOW TYPE	Main lobe Width	Side lobe Amp. (dB)
Rectangular	$4\pi / N$	-13.6
Hamming	$8\pi / N$	-41
Hann	$8\pi / N$	-31
Bartlett	$8\pi / N$	-25
Blackman	$12\pi / N$	-57

Kaiser:

Perhaps the best and most common windowing method however is that of a

Kaiser window [4]. Kaiser created a window that did not have a fixed shape, but rather a

function in which both main lobe width and side lobe amplitude can be optimized. It is

defined as:

$$\omega(k) = I_0\left(\alpha\sqrt{1-(k/N)^2}\right)/I_0(\alpha), \qquad -N \le k \le N$$
$$= 0, \qquad\qquad\qquad \text{otherwise} \qquad (2.29)$$

where I_0 is the zero order modified Bessel function:

$$I_0(\alpha) = 1 + \sum_{k=0}^{\infty}\left[(\alpha/2)^k(1/k!)\right]^2 \qquad (2.30)$$

which can be termed as a power series:

$$I_0(\alpha) = 1 + \alpha^2/4 + \left[(\alpha^2/4)*(\alpha^2/4)*(1/2)^2\right] + \left[(\alpha^4/16)*(\alpha^2/4)*(1/3)^3\right] + ... \quad (2.31)$$

The two parameters of this function affect the window in the following manner

[3,5]:

--Increasing N while keeping α constant will decrease main lobe width without a large side lobe increase

--Adjusting α while keeping constant N will adjusted side lobe level without greatly affecting the main lobe

In order to implement a Kaiser filter, we must first define the Kaiser parameters. We need to specify the parameters of the filter including cutoff frequency (f_c), transition bandwidth (f_t), and ripple (A) [6].

-- f_t must be a fraction of the sampling frequency (usually 2π) and must be less than 0.5. If $f_t > 0.5$, aliasing effects will occur .

--*ripple* is a fraction of pass band amplitude that will not cause significant error and is expressed as an attenuation in dB:

$$A = -20\log_{10}\left(ripple\right) \tag{2.32}$$

For example if ripple is 1% or 0.01

$$A = -20\log_{10}\left(0.01/1\right) = 40dB \tag{2.33}$$

Once this is known, α can be found as [3]:

$$
\begin{aligned}
\alpha &= 0.1102\left(A-8.7\right), & A &\geq 50 \\
\alpha &= 0.5842\left(A-21\right)0.4+0.07886\left(A-21\right), & 21 &< A < 50 \\
\alpha &= 0, & A &\leq 21
\end{aligned}
\tag{2.34}
$$

Finally, to get the number of terms (N) one must use an α value and transition bandwidth (f_t) in the following equation:

$$N \geq (A - 7.95)/(28.72 f_t) \qquad (2.35)$$

Since this tends to give fractional results, the value N should be rounded up to the nearest integer value. If a value of $(2N+1)$ is too long, *ripple* and f_t values can be adjusted [4]. A graphic value representation is shown in Figure 2.5.
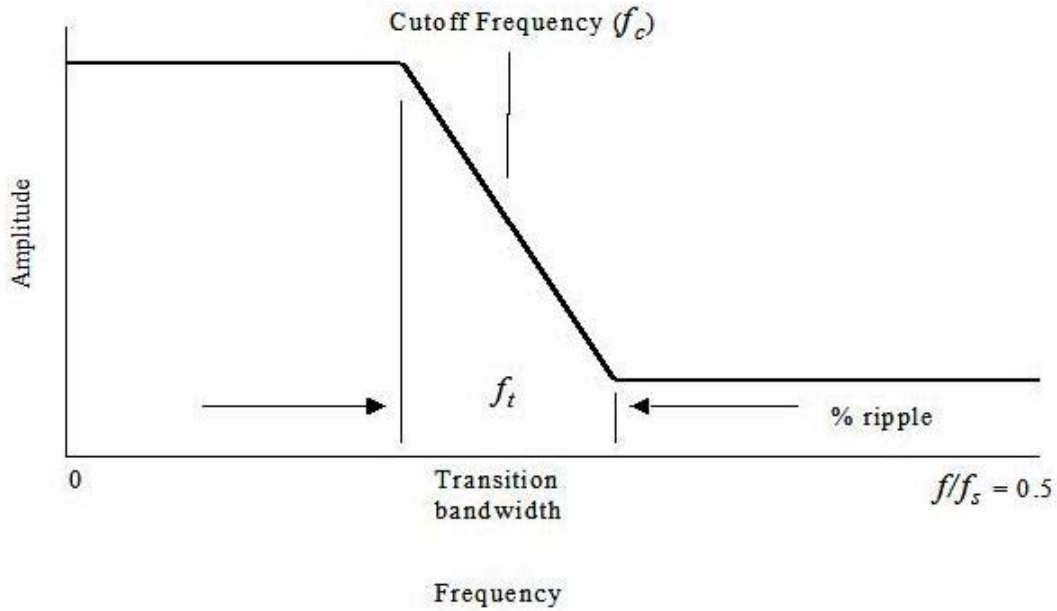
Figure 2.5: To create a Kaiser filter, we must specify cutoff frequency, transition frequency, and percent of ripple allowed [3].

The final step in calculating the Kaiser filter coefficients is to calculate:

$$c(0) = 2f_c$$
$$c(k) = \sin(2\pi k f_c)/(k\pi) * \omega(k)$$

(2.36)

where k is obviously ranging from 1 to N [6].

2.5 Alternate Methods and Studies of I/Q Down-Conversion

There have been, and are currently a number of research projects that have been dedicated to creating I and Q down-converters. Before A/D converters became fast enough to handle higher-bandwidth waveforms, the I/Q demodulation was done using an analog system such as the one shown in Figure 2.6. Now that A/D converters that can operate in the GHz range are emerging as a viable strategy however, Digital Down-Converters (DDCs) have become the subject of much research and development. Due to remaining digital limitations, some of these implementations have used methods that may cause unnecessary size and power to be added to the circuit. A couple of the alternative methods that are being used to create digital I/Q downconverters will be discussed.
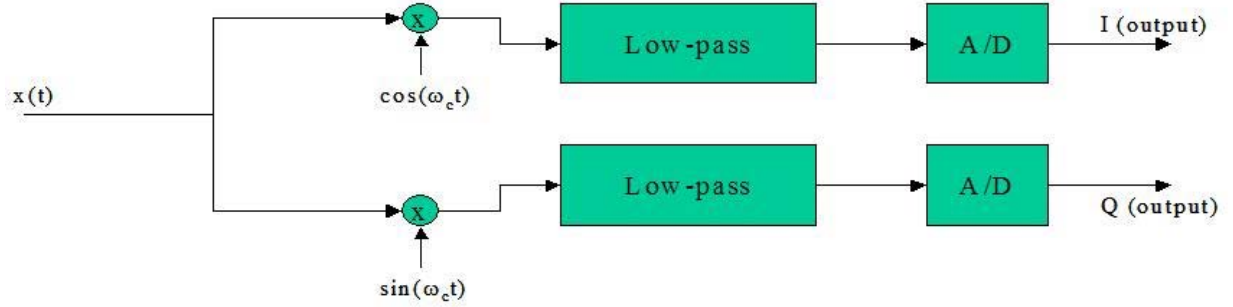
Figure 2.6: Analog I/Q demodulation followed by A/D conversions [7].

Virtually all of the current DDC's use a method of sampling that utilizes a Hilbert transform. According to Tsui [2], a Hilbert transform that is applied to a function *x(t)* in the time domain can be defined as:

$$H[x(t)] = x(t) \otimes h(t) = x(t) \otimes \frac{1}{\pi t} \tag{2.37}$$

where once again the $\otimes$ represents a convolution, and $H[x(t)]$ represents a Hilbert transform. A result of a Hilbert transform can also be found using an inverse Fourier transform such as:

$$H[x(t)] = x(t) \otimes h(t) = F^{-1}[X(f)H(f)] \tag{2.38}$$

40

The Fourier transform of $h(t)$, which represents $H(f)$ above, can be represented as:

$$F[h(t)] \equiv H(f) = j\,\mathrm{sgn}(f) = \begin{pmatrix} -j & f > 0 \\ j & f < 0 \end{pmatrix} \qquad (2.39)$$

Equation (2.39) above leads us to conclude that a Hilbert transform only has an imaginary part H_i, and therefore the real component H_r can be ignored. These equations are expanded for better understanding in [2, Chap 8]. Also in [2, Chap 8], a simple example is used to demonstrate the effect of a Hilbert transform. Taking an input signal of:

$$x(t) = \sin(2\pi f_i t) \qquad (2.40)$$

which has a Fourier transform of:

$$X(f) = \frac{j}{2}[\delta(f + f_i) - \delta(f - f_i)] \qquad (2.41)$$

where $\delta(\ )$ is the dirac function at a specific frequency.

Since the Fourier transform of $h(t)$ allows us to multiply Equation (2.40) by j when $f < 0$ giving $-0.5\delta(f+f_i)$, and by $-j$ when $f > 0$ giving $-0.5\delta(f-f_i)$, we can recognize the result as the Fourier transform of $-\cos(2\pi f_i t)$. Comparing the resulting signal to the original in Equation (2.39), we can then see that the result is an equivalent amplitude signal that is $90°$ out of phase from the original. Therefore, the following relationships can be inferred:

$$H[\sin(2\pi f_i t)] = -\cos(2\pi f_i t)$$
$$H[\cos(2\pi f_i t)] = \sin(2\pi f_i t)$$

$$(2.42)$$

The above equation set gives a good example of an analog Hilbert application. Since a discrete Fourier transform is periodic however, that fact will affect the discrete Hilbert transform. Again, Tsui [2, Chap 8] gives us a good analysis of a discrete Hilbert transform manipulation in which he takes into account the needed causality of digital sampling. His Fourier series representation of the Hilbert transfer function, assuming a sampling frequency f_s is:

$$H_i(e^{j2\pi f t_s}) = \sum_{n=1}^{\infty} b_n \sin(2\pi n f t_s) \qquad (2.43)$$

where:

$$b_n = \frac{2}{f_s} \int_{-f_s/2}^{f_s/2} H_i(e^{j2\pi f t_s}) \sin(2\pi n f t_s) df$$

$$(2.44)$$

$$= \left(\begin{array}{ll} 0 & n = \text{even} \\ \dfrac{-4}{n\pi} & n = \text{odd} \end{array} \right)$$

using Tsui's derived relationship of:

$$h(n t_s) = -\frac{b_n}{2} \qquad (2.45)$$

and applying to equation (2.43) in a discrete, causal manner, we find that:

$$H_i(e^{j2\pi nft_s}) = -2\sum_{n=1}^{\infty} h(nt_s)\sin(2\pi nft_s) \qquad (2.46)$$

$$h(nt_s) = \begin{pmatrix} 0 & n = \text{even} \\ \dfrac{2}{n\pi} & n = \text{odd} \end{pmatrix}$$

$$\qquad (2.47)$$

$$h(-nt_s) = \begin{pmatrix} 0 & n = \text{even} \\ \dfrac{-2}{n\pi} & n = \text{odd} \end{pmatrix}$$

Finally, using the values from Equation (2.47), we can create a set of coefficients for a Hilbert transform filter. Using a FIR with a rectangular window and 11-taps, or coefficients, we get $h(n)$ values corresponding to n values from –5 to +5. These values and corresponding causal values are demonstrated in Table 2.2. We could also use an additional window with coefficient values $w(n)$ by multiplying $w(n)$ to $h(n)$ in the Hilbert transform filter. Such windows as those highlighted in section 2.4 are often used to limit frequency effects of using a simple rectangular window.

Table 2.2: Standard $h(n)$ values of a Hilbert transform filter.

n (non-causal)	-5	-4	-3	-2	-1	0	1	2	3	4	5
h(n)	$\dfrac{-2}{5\pi}$	0	$\dfrac{-2}{3\pi}$	0	$\dfrac{-2}{\pi}$	0	$\dfrac{2}{\pi}$	0	$\dfrac{2}{3\pi}$	0	$\dfrac{2}{5\pi}$
New n (causal)	0	1	2	3	4	5	6	7	8	9	10

Current research studies are being performed by students at Wright State University (WSU) under the supervision of Dr Marty Emmert in collaboration with the Air Force Institute of Technology (AFIT) and AFRL, to make use of both the SSS

method and the Hilbert transform. One such Hilbert transform project is a parallel of this

study in producing I and Q waveforms for radar processing by using high-speed serial

input data. Since both methods of I/Q digital data conversions are under-developed, this

collaboration will serve an integral part in the maturing and development of these

receiver technologies in the future [8].

Sjostrom et. al. also made prevalent use of a discrete Hilbert transform in the

design of a DDC chip for use with an S-band digital beamforming array antenna in 1996

[9]. Their aim was similar to this study in that they were trying to minimize size, power,

and data imbalances. They used a 12-bit ADC input in which they split in the time

domain and used a Hilbert filter to form I/Q channels. They then used decimation filters

to slow down the signal as shown in Figure 2.7. Their architecture concepts are very

flexible, but there was no abstract top-level HDL code supporting it. They also use a

channel equalizer to mechanically minimize the amplitude and phase differences between

I/Q channels. While this is a valid and adaptable method towards minimal channel

imbalances, which could be used in our design also, it adds another stage of complexity

and along with it, extra size and power consumption to the processor (upwards of 25% or

greater addition to the chip size). This size addition can be seen by looking at their chip

layout in Figure 2.8. Therefore we will not include a channel equalizer in this research.
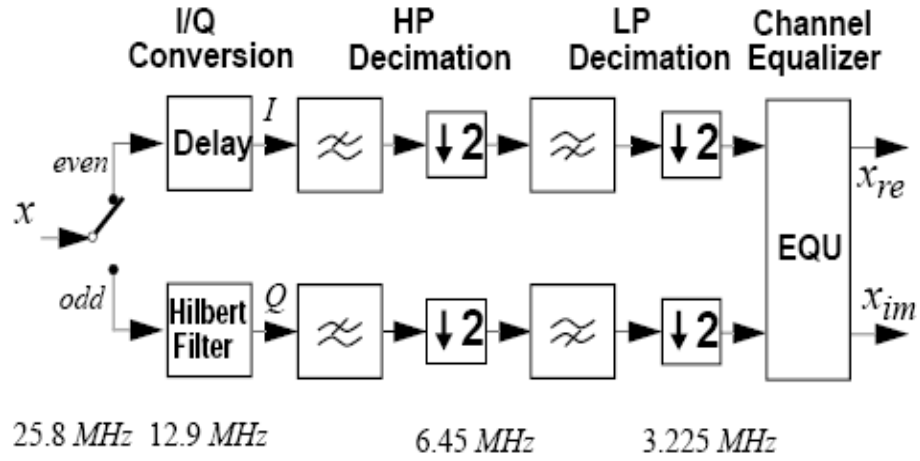
Figure 2.7: Sjostrom et al. digital filter block diagram [9].
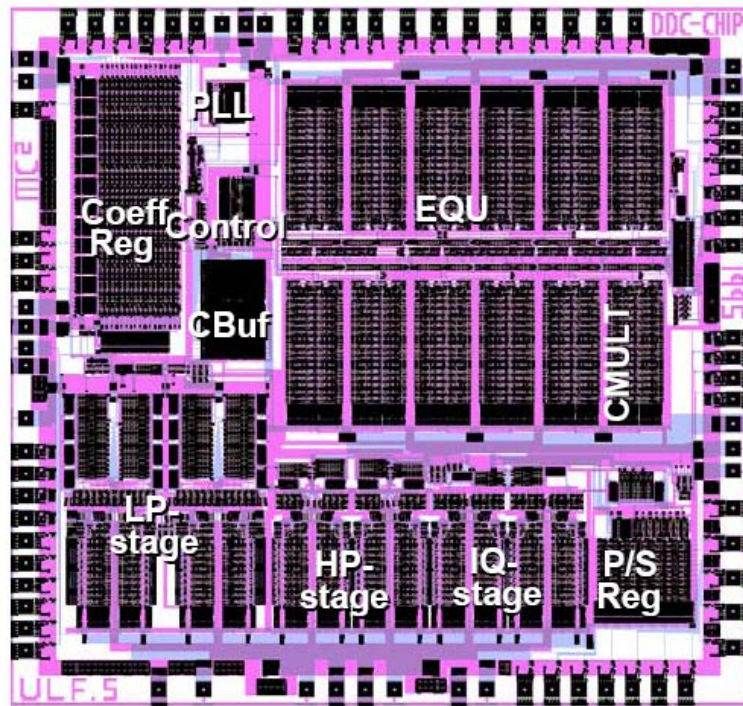
Figure 2.8: Sjostrom et al. digital down converter chip layout [9].

2.6 Spurious Responses in Digital Sampling

In this section, the specific sampling principle of a spurious response will be outlined, which we will use in order to verify and classify the created and manipulated data samples. Spurious responses are vital to this research as they will mark the points of comparison between our I and Q channels of data. We will look at amplitude and phases of these points to determine amplitude and phase imbalances between the channels.

In any given sampling of a periodic signal (sine or cosine), we have three main components that will determine the locations of what are called spurious responses [8]: the input signal frequency (f_i), the sampling frequency (f_s), and the sample size, or number of samples taken (S_{size}). These three components help to comprise what is called a frequency spectral bin which is described in detail below.

Let us take for example a signal of frequency 5 MHz that is sampled 100 times, or rather has a sampling frequency of 100 MHz. This data set will form a set of spectral bins that will contain signal energy. The greatest concentration however will spike in the bin corresponding to the input frequency. We can see in Figure 2.9 that when we take an input signal of 5 MHz, sampled at 100 MHz, and sampled 100 times; running it through a Fast Fourier Transform (FFT) gives 100 spectral bins with spurs at 6 and 96 (5+1 and 95+1, due to my specific Matlab charting technique, the plots are shifted slightly). Since the input frequency exactly matches one of the sampled frequency bins, and the sample data accuracy is close to exact, there are no side lobes in the plot. If we were sampling a frequency such as 2.5 MHz or 8.4 MHz, it would not match exactly with a spectral bin and would therefore form side lobes in the plot, creating many erroneous spurs. This concept is demonstrated in Figure 2.12 for quantized data. Another interesting

46

consistency in a spectral bin plot is the symmetrical mirror effect that occurs. We can see that points 2 to 51 are a mirror of those from 52 to 101. This will allow us to only need analysis of one half of the plot for certain data.
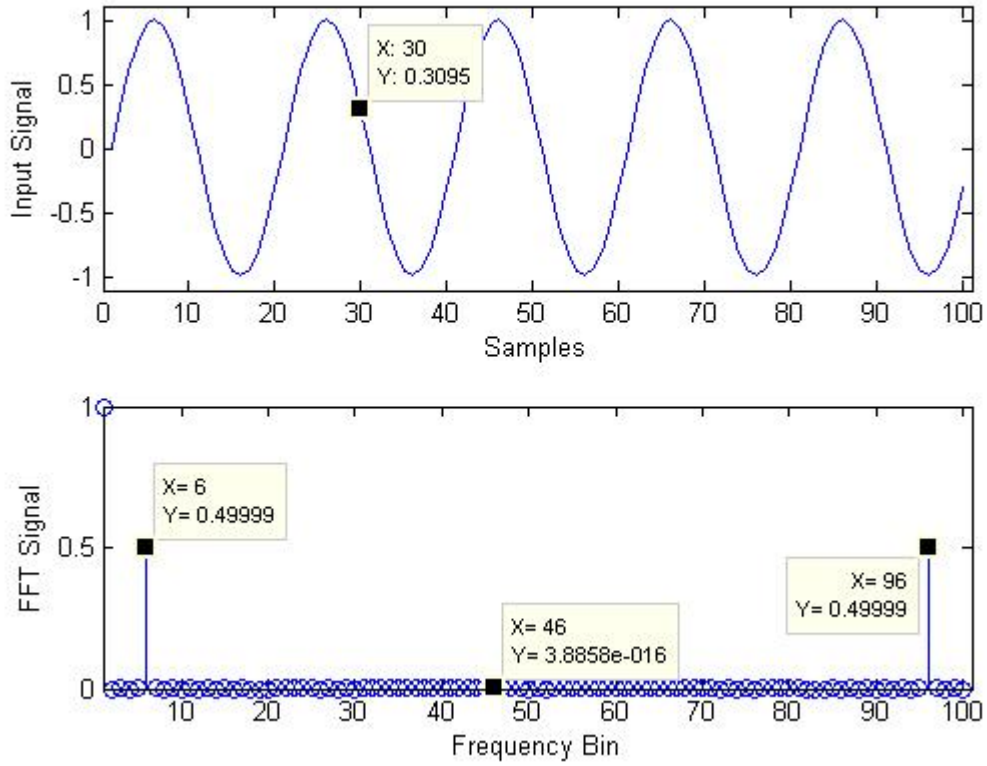
Figure 2.9: Plot of $f_i = 5$ MHz, $f_s = 100$ MHz, S_{size} of 100 and non-quantized data.

Now if we were to quantize the input data set to a binary from, and truncate it to a signed form with 3 bits, we will get a new spectral bin with erroneous spurs. These slight side lobe spurs are a result of the loss of accuracy in the data points due to the truncation. Such a plot is shown in Figure 2.10. The error between time domain signals of Figure 2.9 and Figure 2.10 is shown in Figure 2.11. We can see that the errors repeat themselves each cycle and will therefore continue to repeat if more samples are taken.

47
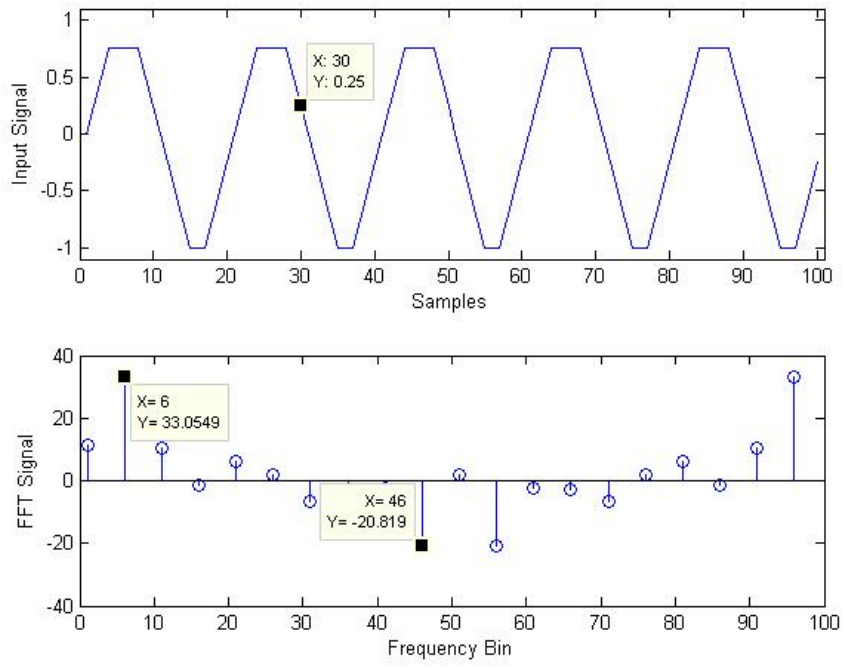
Figure 2.10: Plot of f_i = 5 MHz, f_s = 100 MHz, S_{size} of 100 and quantized data.
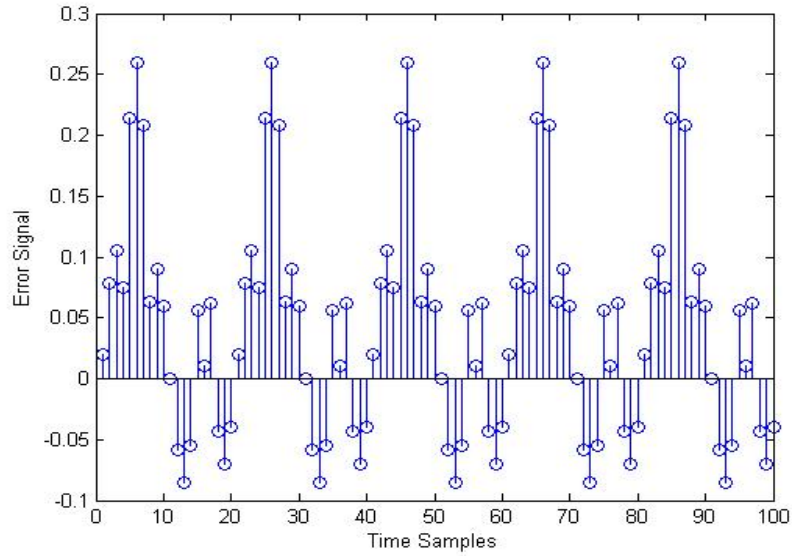
Figure 2.11: Difference between quantized (3-bits) and non-quantized data.

When we take a sample as stated above in which the input frequency does not match exactly to a spectral bin, we see much higher side lobes than if just using truncation. In Figure 2.12, we again sample at 100 MHz with a 3-bit representation, but this time we will use an input frequency of 2.5 MHz. Here we have side lobe spurs of a significant amplitude and thus have to be careful in this case when analyzing data samples. This affect can be especially important if reading data containing more than one frequency component.
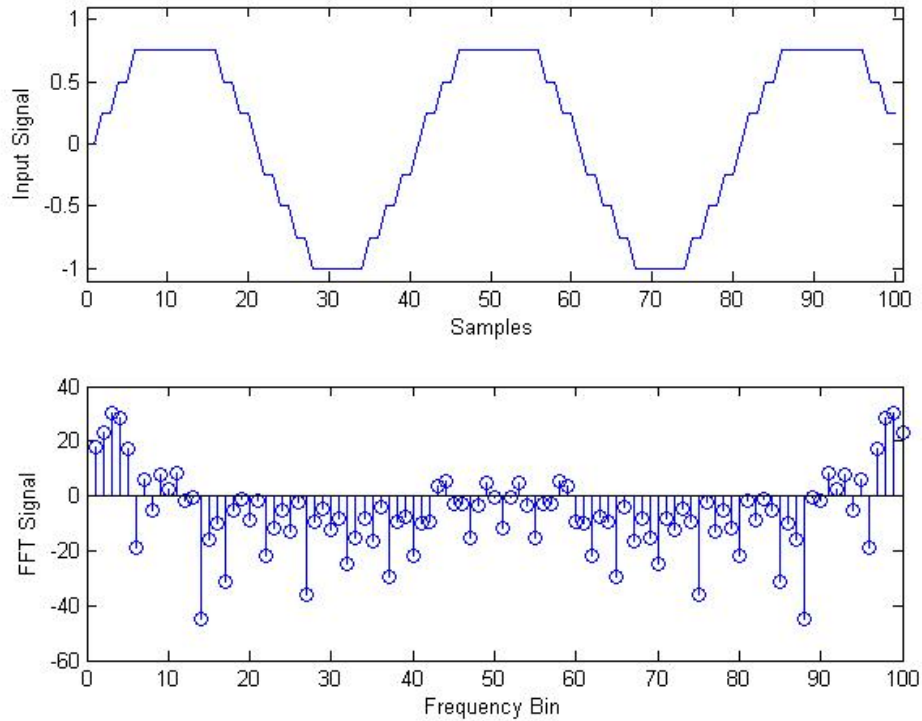
Figure 2.12: Plot of $f_i = 2.5$ MHz, $f_s = 100$ MHz, S_{size} of 200 quantized data points.

To show what will happen if the number of samples is increased, we can run the same sample as in Figure 2.10 with 6-bit quantized data, but this time use 200 samples as in Figure 2.13. Here we see basically the same plots as in Figure 2.10; however, we have used twice as many periods of the input wave, and therefore the spectral bin is also proportionally halved in bin-width, and bin amplitudes are also proportionally increased. In general, we can assume that the more spectral bins there are in the sample, the more concentrated the spurious response will be. The frequency bin of the ideal spur can be represented as:

$$B_{max} = \frac{S_{size}}{f_s} * f_i \tag{2.48}$$

where B_{max} is the bin number of the ideal maximum spur, S_{size} is the sample size, f_s is the sampling frequency, and f_i is the frequency of the input signal. In the Figure 2.13 we can see that:

$$B_{max} = \frac{200}{100} * 5 = 10 \tag{2.49}$$

therefore the ideal spur bin should be located at about 10 (+1 for my scheme: = 11).
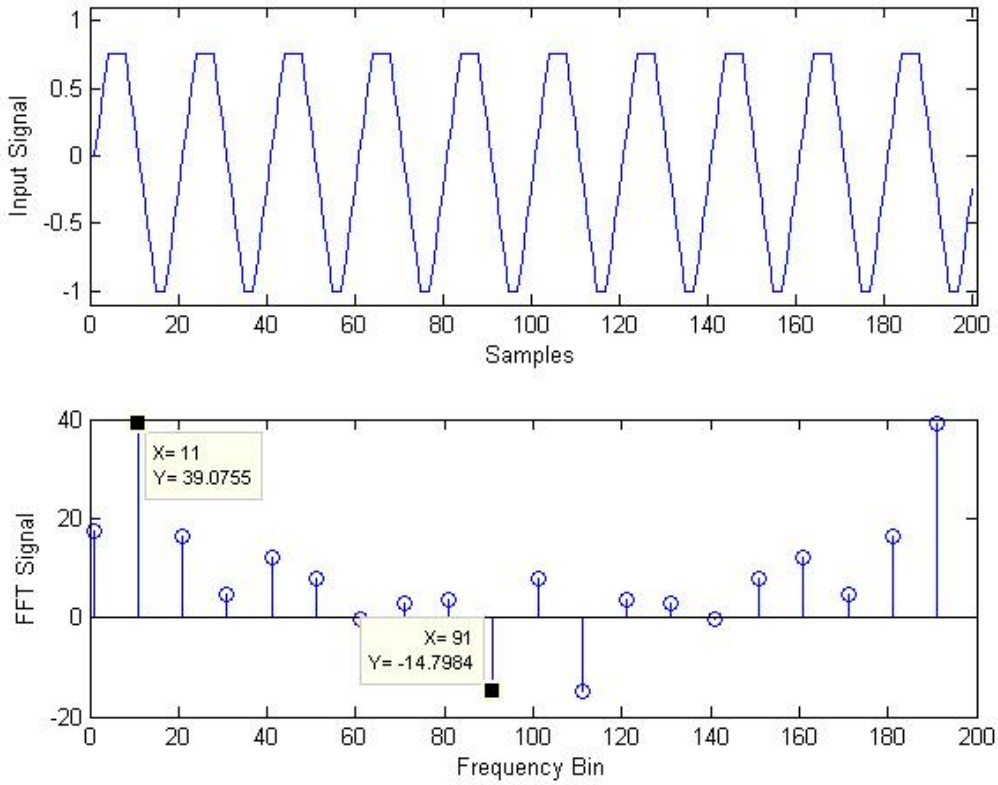
Figure 2.13: Plot of f_i = 5 MHz, f_s = 100 MHz, S_{size} of 200 quantized data points.

An important detail in developing all plots in this research is that the input signal must start at $t = 1*10^{-8}$ and continue to $t = 4\pi + 1*10^{-8}$. This is required because if the signal were to pass $t = 0$, computer errors due to the zero will greatly affect the quantized data sample [2, Chap 6].

2.7 Spurious Signal Analysis

Now that we know how the spurious signals are derived, we need to apply the concept in order to get useful information for classification of I/Q imbalances. Every signal that is run through an FFT function, will contain both an imaginary (Im) and a real

(Re) component, and therefore every spectral bin will contain a Re and Im component.

Using this concept allows us to represent each spectral bin in a polar notation with a

radius (R) and angle (θ) [8]. A circle representation can be used to visually present this.

If we use the y-axis to represent the Im component and the x-axis to represent the Re

component, we can get points on a circle that will represent a radius measure from the

origin and an angle measure from the positive x-axis. For example, if we have a Re

component of value 0.8660254 and an Im component of value 0.5, it will be placed on

the circle representation as in Figure 2.14.
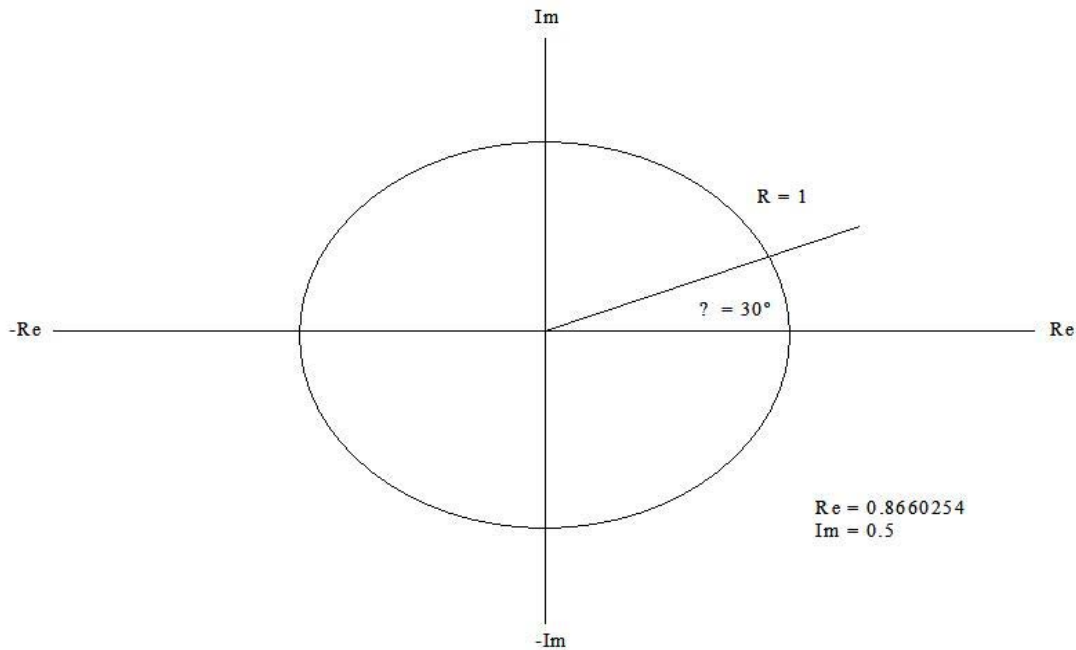
Figure 2.14: A polar (circular) representation of a signal with amplitude of 1.0 and a
phase of $30°$.

We can see above that the values form a line on the circle and that if we use the

following equations, we can get the amplitude (or radius $\{R\}$), of the line and the phase

(or angle $\{\theta\}$) of the data point:

$$R = \sqrt{\text{Re}^2 + \text{Im}^2} = 1$$
$$\theta = \arctan \frac{\text{Im}}{\text{Re}} = 30°$$

<div align="right">(2.50)</div>

Each spectral bin of an FFT signal will contain an amplitude and phase as above, but to get the most accurate representation of the amplitude and phase of the entire signal, we want to measure the components in the bin where the signal has it's maximum spur. This maximum spur can be found easily by comparing the amplitude of each bin. If side lobe and erroneous spurs in the signal are kept to a minimum, this method is quite efficient for calculating imbalances of the entire signal.

For our purposes, we can run an equivalent FFT function on each of the I and Q data signals sampled from the original input signal. Thus by doing so, we will then have an accurate polar model of each signal that will make finding imbalances between the data trivial.

Because the I and Q channels should have the same frequency, the maximum spur should occur at the same spot in each of their FFT outputs. The FFT data from each channel should ideally have equal amplitudes and will be $90°$ out of phase (the I channel most commonly leads the Q channel). By comparing the data at this spot for each channel, we can clearly see the imbalances between channels. Figure 2.15 shows a second channel on the same plot as the one in Figure 2.14. This is an ideal representation as we see that the I channel and Q channel both have amplitudes of 1.0 and that the I channel leads the Q channel by exactly $90°$.
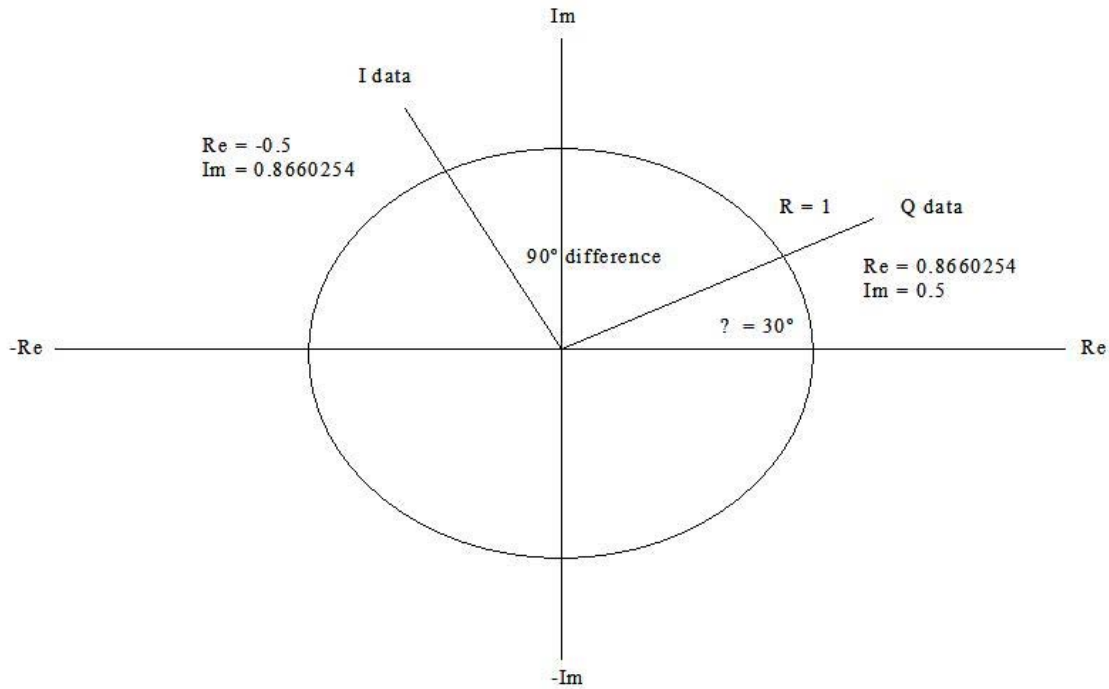
Figure 2.15: A polar (circular) representation of two signals with amplitudes of 1.0 and phases of $30°$ and $120°$ ($90°$ out of phase).

2.8 Design Flow of Digital Synthesis

In previous sections we have defined all of the background knowledge needed to understand the SSS and how it will be characterized. That information will help us to create the VHDL models of the SSS circuit, but it does not explain any way to validate the design. In this section we will take an in depth look at the background required to understand the design synthesis process, and optimization of Hardware Description Languages (HDLs).

Modern day circuit design has led to a number of trends and patterns. Such factors as ever-growing complexity and density, ever shrinking sizes, and always growing need for custom design, have influenced the design process for ASICs,

microprocessors and Systems-on-Chip (SoCs) in many vital ways. The nano-scale of

modern design implementations brings about ever increasing problems of routing circuits

while still maintaining timing and congestion standards on the chip. Now that we have

met 130 nm technology and are heading into the 90 nm and even 60 nm era, second and

third order physical effects are posing difficult barriers that went unaccounted for in the

past [10].

The front-end of ASIC design now yields to the back-end as physical effects of

implementation are now slowing logical designs more significantly. Gate delays used to

be a major factor in circuit designs, but as can be seen in Figure 2.16, the largest slow

down today is due to interconnects and routing delays. Because of this, design iterations

and design flows are becoming absolutely vital to the implementation and optimization of

such circuits. Industry shows a great need for convenient and fast design models so that

the hand-off of tape-outs, or finalized design files to be sent to foundries, can be

accomplished with less design iteration. Government agencies such as the military on the

other hand are interested in building fully optimized designs that can compete with

industry standards. Algorithms that combine both logical circuit verification with that of

'look-ahead' synthesis checks (router, timing, power, etc.), are of great use to both

industry and government circuit engineers and designers.
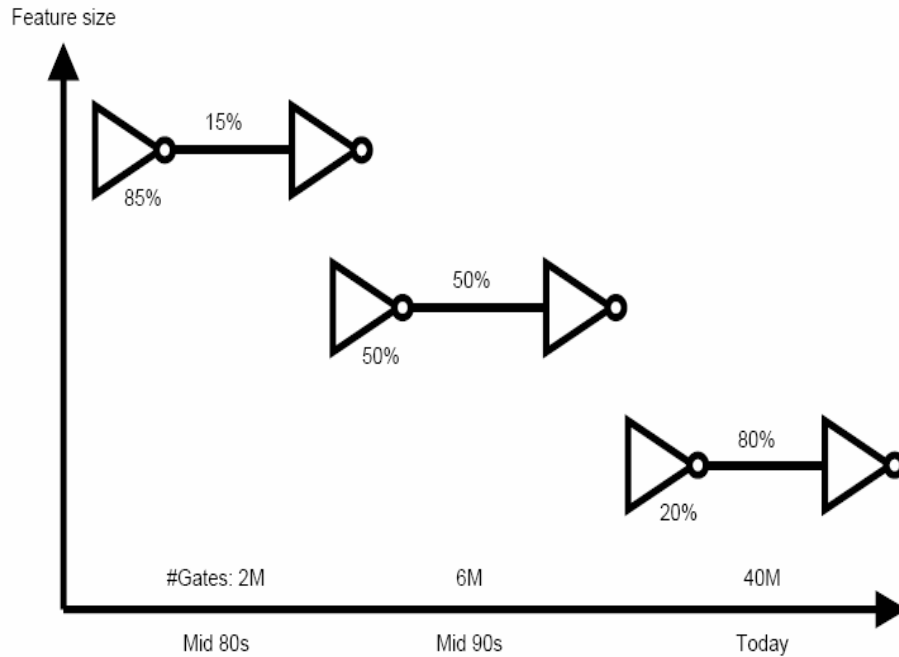
Figure 2.16: Evolution of gate delay contribution to interconnect delay, as feature size decreases and die size expands [10].

The most basic (classical) digital circuit design flows seem to involve two major design steps, with a major verification/optimization step following each [11]. These design steps are finalized with a final tape-out step. Each step and verification process has its own challenges and variations, and involve a number of smaller processes of its own. A description of each section of an ASIC circuit design flow is outlined as follows:

-<u>Top Level Design:</u> The first step in the development of a circuit, often termed the 'front end' is the initial design and top-level implementation. Another way to look at this step is as an architectural draft. This series of work is inclusive of the brainstorming and requirement gathering steps, and the final top-level design outline. Decisions made in this work include evaluating design feasibility, determining if the design will fit on a

specified die, and determining if it will meet the required timing and power constraints. Once a plan has been made for the circuit, top-level block designs can be created using pre-existing libraries of cells, or by creating custom cells with known or designer-created macros [12].

At one point, this initial design step was accomplished at a low transistor level known as 'structural.' However, with the ongoing and continuous nature of technology to go smaller in size, but bigger and more complicated in scope (high density and number of transistors—multimillion gate designs are common place), traditional low level models no longer suffice as the computer resources are not available in most cases to handle such designs.

Instead, designers have turned to the top-level, 'behavioral' design methods known as Resistor-Transistor Logic (RTL) [10]. Industry and educational institute standard HDLs such as VHDL and Verilog have put this logic to use. Using the programming syntax in these languages, designers are able to easily program a description of their device and then use a simulation or synthesis tool to extract that description into an actual device cell. There are many industry standard cell (macro) libraries available for use, or designers can choose to customize their own. Once a 'behavioral' design has been created by combining one or more series of macros it needs to be verified and tested for functionality.

-Top Level Verification: Optimization within the top-level design comes from using Electronic Design Automation (EDA) tools available to today's designers. Industry leaders in this design phase such as Synopsys, Cadence, and MentorGraphics all supply

designers with programs that allow combination of a controlled level of detail in design, while still ensuring high-level viability in silicon.

Initially, a basic company package will have software to deal with VHDL or Verilog programming and code. Through these a designer can write a code, and then verifying it along the way to ensure it will compile, therefore avoiding some simple syntax errors that can waste a designer's time if not found early on. Coding softwares also commonly include simulation runs that will allow verification of architectures of simple blocks (such as a mux or adder) or of larger systems like calculators or clocked units. Such software is vital in verifying the first designs for actual functionality.

The next part to optimizing the design comes in turning such 'logical' elements of a VHDL design into physical cells that can be used in a fabrication. Again, this is covered by industry programs that use Design Rule Checks (DRCs) and automatic routers to automate this RTL-to-gates transition [10]. This is a process that will take the 'logic' and convert it using rules of the sized technology of the design, into metal wire layers, in the most optimized way the program can. Most often, a large design will be made into cells (mux, adders, etc.) that are combined for a final result. These cells are optimized to be as small or efficient as possible. Cells can be automated (such as reuse of IP blocks), or can be designed manually by a user with programs like Magic and Cellgen. It would be most effective to take a whole design into a DRC and have it implemented into one single routed design, but in the large systems being created today, such a method would be much too complex to be handled by CPU overhead available.

For the reasons stated above, as well as for flexibility needs of designers to adjust a design to certain optimizations (optimizing a design for speed, will result in a different

58

design than if altering for other factors such as size, power, or reliability), the approach used currently is to start with a 'cell' design, and alter the way in which the cells themselves are placed and routed to one another [13]. This is a vital step to a design process in which a great number of iterations may occur. Placement optimization software allows a user to do multiple runs of placements options such as ratio, rotation and mirroring, and to then estimate a die size that will be needed. A great advantage to iterating at this point is that blocks can be added or removed to create different variations of the design, which can be run under cost analysis.

The large advantage to having an optimized front-end design is to give gate-level physical tools the best possible starting point later on in the process. Such optimization tools however cannot suffice for the physical gate-level because prototyping placement tools have extra options such as 'bleed out' (where gates can extend out of their block and into open areas of neighboring blocks) [10]. A good starting point for the physical level placement is essential however for saving time in back-to-front iterations in design, and can help bring much earlier timing closures. A good block diagram representation of a front-end process is shown in Figure 2.17.
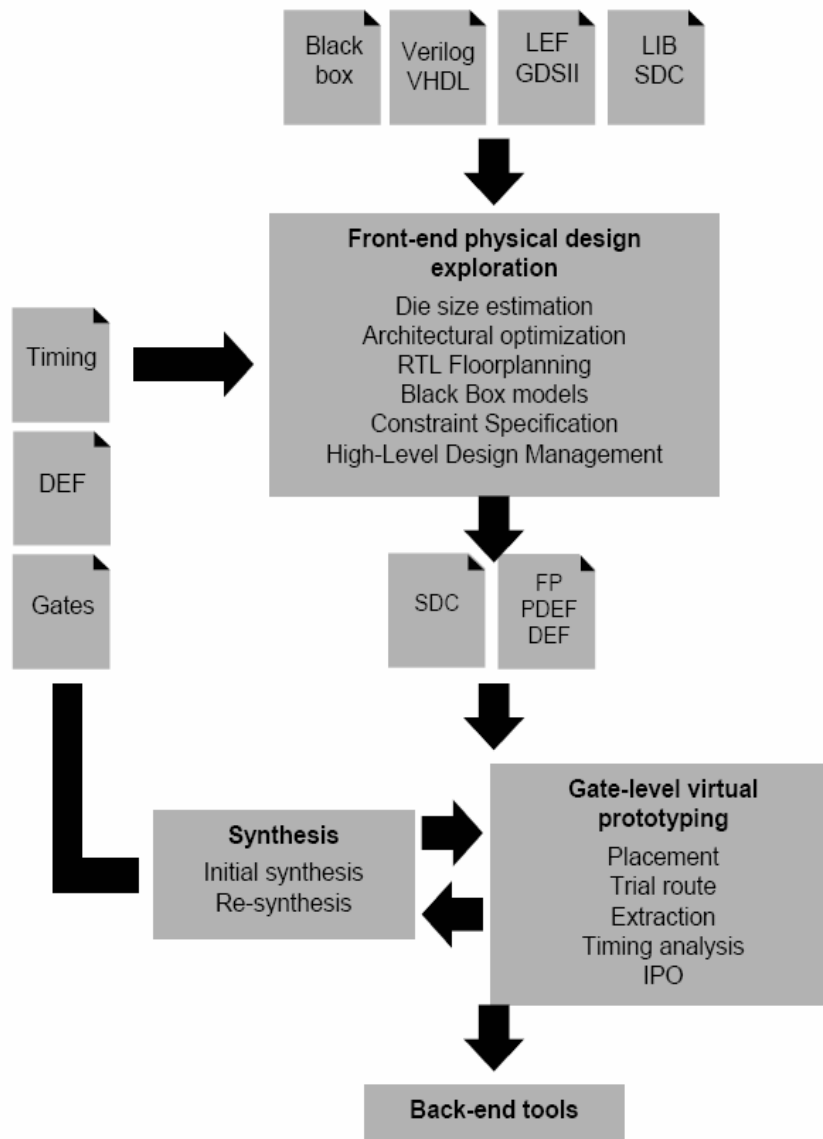
Figure 2.17: Front-end physical design exploration flow into synthesis and virtual prototyping tools. [10]

-Physical Design: Once an initial design has been decided on in terms of rough size and placement, it must be synthesized for test in a physical environment. This part of the process is often referred to as the 'back-end' of the design process. This portion of the design process usually does not result in large changes in the physical makeup of the

circuit, but it does make adjustments so that the design may be implemented with the technology specified (dependent on size: submicron, Deep-Submicron (DSM), and Very-Deep-Submicron (VDSM)).

   As technologies go down in size, the impacts of electrical interconnects create increasing difficulties in sustaining timing and routing consistency, as well as sustaining power issues [14]. If timing closure cannot be achieved by the design as it is, it cannot be successfully fabricated to working order. Therefore, the physical implementation of an ASIC design is critically important to adjust until correct, otherwise an actual fabrication of the design may prove dysfunctional.

   -Physical Verification: Optimization within the back-end of a design starts with extraction of the front-end design into a physical design-optimizing program. The goal of this is to make up for inaccuracies of wire load models by implementing Resistor-Capacitor (RC) effects. By accounting for these new parameters, designs can be optimized with new models and non-necessary (detour) routes due to congestion within the wiring can be avoided. Timing closure can then be accomplished in conjunction with such physical effects as blockages, clock tree synthesis, scan chain re-ordering among other low-level details not accounted for in the high-level design process [14]. This being the case it is easy to see why front-end optimization is important to give a good starting point, but is unable to give a sufficient final design.

   The first and foremost element in back-end design is placement. It is critical because it affects routability, reliability, and die area, and therefore has a huge impact on a design meeting its timing specifications. Just a small change to the placement of design

elements to optimize one physical effect, can have a serious impact on the optimization of others. A significant movement in place of elements will affect RC calculations that are used to determine gate size and logic order. In DSM circuits, multiple objectives such as routability, timing, and signal integrity can be designed around (with good algorithms, more than one can be optimized at a time in a balanced effort) [14]. To manage a complex design efficiently, a good tool needs to be able to identify true timing paths, as false and looping paths will waste logical efforts, hide underlying timing problems, and cause congestion.

As technology continues to shrink, effects of interconnect on delay will increase. It is then important for a physical optimizer to detect so called 'hot spots' and modify the wires to relieve congestion [14]. If congestion is kept, there will be discrepancies in routing behavior and timing closure will be difficult due to created detour routes of wires. False paths and detoured routes cause extra time to be spent on analysis.

There are a number of other results of wire interconnect in an ASIC design. Pessimistic wire load models can cause excess buffers to be routed into the design and create congested areas, as well as take up extra power and area. A second issue is that of set-up and hold times. Varying gate size or adding appropriate buffers can fix violations of set-up or hold times. A third very integral issue to be managed is signal integrity, which may include crosstalk, noise, and power. There are a number of ways to handle all of these elements [14].

2.9 Chapter Summary

In this chapter, we clarified the background information needed to understand the various parts of creating a high-level VHDL I/Q modulator unit using the SSS. Such factors as digital representations and filtering methods, along with actual methods of sampling incoming digital data was reviewed. Also covered was the process of design synthesis to tape-out of a working ASIC circuit.

2.10 References

[1]   "Modular Digital Radio Frequency System (MoDRFS)." Final Report, Raytheon Company. Government Contract: F33615-98-2-1330. 19 May, 2004.

[2]   Tsui, James. *Digital Techniques for Wideband Receivers.* Artech House, 2001.

[3]   Zelniker, Glenn. and Taylor, Fred J. *Advanced Digital Signal Processing.* New York: Marcel Dekker, Inc, 1994.

[4]   Lynn, Paul A. and Fuerst, Wolfgang. *Introductory Digital Signal Processing With Computer Application.* John Wiley & Sons Ltd., 1994.

[5]   Oppenheim, Alan V. and Schafer, Ronald W. *Discreet-Time Signal Processing.* New Jersey: Prentice-Hall, Inc, 1999.

[6]   Haussman, Werner. "Filter Your Data in Software." *Test & Measurement World.* 12 October 2005 http://www.reed-electronics.com/tmworld/index.asp?layout=articlePrint&articleID=CA202503.

[7]   Shaw, Gary A. and Pohlig, Steven C. "I/Q Baseband Demodulation in the RASSP SAR Benchmark." M.I.T. Lincoln Laboratory. 08 November 2005 http://www.ll.mit.edu/llrassp/I/Qfilter/I/Qfilter.html.

[8]   Personal discussion with Col Marty Emmert of the USAF AFRL and Professor at WSU. November 2005.

[9]   Sjostrom, Ulf. and Carlsson, Magnus and Horlin, Magnus. "Design and Implementation of a Digital Down Converter Chip." *The European Association for Signal Image Processing.* 04 November 2005 http://www.eurasip.org/content/Eusipco/1996/paper/i_7.pdf.

[10]   Clee, Christopher and Lee, Suk. "The Effect of Nanometer Technology on ASIC Handoff." White Paper, Icinergy Software Company & Cadence Design Systems. 17 January 2006 http://www.cadence.com/whitepapers/icinergy_cadence_whitepaper_final.pdf.

[11]   "Digital Custom Circuit RTL Model Development and Functional Verification." Technical Paper, Cadence Design Systems. 17 January 2006 http://www.cadence.co.in/whitepapers/equivalence_checking_WP.pdf.

[12]   Personal discussion with Dr. Yong Kim, Professor at AFIT. August 2005.

[13]   Idgunji, Sachin., et all. "Design Planning Strategies to Improve Physical Design Flows—Floorplanning and Power Planning." White Paper, Synopsys, Inc. August 2003.

[14]   Wilson, Jeff. "Achieving DSM Timing Closure with TeraPlace, a Physical Optimization Solution." White Paper, Mentor Graphics Corporation. 2000.

III. Characterization & Methodology

3.1 Bit Precision in Digital Modeling

This section, and the following sections of this chapter, will outline the principle strategies to be used in the characterization and implementation of a SSS for I/Q downconversion utilizing top-level, portable design strategies. This specific section will review and demonstrate the importance of using binary representation in our characterization models and will outline the representation options we have available.

Because decimal notation is not a hardware option, it is critical to define a number representation before moving into characterizing signals and creating circuit designs. Since Matlab, by default, uses a decimal system of representing numbers, the numbers must be altered in order to create a binary representation of the numbers. Hardware implementation must use simple math functions and limited bit-widths in order to obtain high speed processing (normally due to ADC limitations). AFRL's fastest ADC can operate near 3 GHz; however as a result, it also is limited to only two bits in width which makes it impractical for most uses [1].

One of the biggest debates in Digital Signal Processing (DSP) is whether to use a floating-point representation, or a fixed-point representation in a design. Floating-point operations in general, contain more logic levels than do fixed-point operations in an ASIC or FPGA design. This usually results in a greater clock-cycle cost [2]. Historically, to get real-time performance, it has been required to primarily, if not completely, use a fixed-point design. The trade-off to this however, is a greater loss in numeric fidelity of the processed information over that of a floating-point operation.

66

There are two IEEE standards for Floating-Point precision of a signed number: 32-bit and 64-bit. This is called the IEEE 753 *floating-point* standard, and had been used in computers and devices since 1980 [3, Chap 3]. The 32-bit binary floating-point representation can be seen in Table 3.1 below, in which the first bit is the sign bit ('0' for positive and '1' for negative), the next eight bits make up the exponent value, and the last 23-bits, referred to as the mantissa are the fractional values. The 64-bit binary floating-point representation (referred to as *double precision*) in Table 3.1 is the same format with a sign of 1-bit again, an exponent of 11-bits, and fraction of 52-bits.

Table 3.1: IEEE 753 floating-point representations.

IEEE 753 Floating-point	Sign	Exponent	Fraction (Mantissa)
Standard	1-bit	8-bits	23-bits
Double	1-bit	11-bits	52-bits

When reading a floating-point value, one may convert to its decimal equivalent by using the equation:

$$D_{value} = (-1)^s * (1 + Fraction) * 2^{(Exponent - Bias)}$$

(3.1)

where *s* is the sign bit, and both the *Fraction* and *Exponent* are the decimal equivalents of their binary values. The *Bias* is needed to correctly distinguish positive exponent values from negative exponent values and had a value of 127 for single precision, and 1023 for double precision. To demonstrate, let us use a binary number represented in single precision:

67

$$B_{value} = 1 \mid 10000001 \mid 01000000000000000000000$$

using the floating-point terms we can extract:

$$s = 1$$
$$Exponent = 10000001 = 129$$
$$Fraction = 01000.... = 0.25$$

and therefore:

$$D_{value} = (-1)^1 * (1 + 0.25) * 2^{(129-127)}$$
$$= -1 * 1.25 * 2^2$$
$$= -1.25 * 4$$
$$= -5.0$$

Floating-point additions and multiplications require fairly involved algorithms to solve equations. Most of the complexity of the operations involves steps to normalize the inputs and the results to match each other's fractional decimal point places. A full explanation of the algorithms can be found in [3, pg197-205]. Hardware logic examples of a floating-point adder and multiplier can be seen in **Figure 3.1** and Figure 3.2 respectively.

In addition to the standard IEEE floating-point structures, lightweight floating-point structures can be developed. The advantage of this is that unnecessary precision can be cut out in exchange for smaller area and power, and in most cases, higher speed. A lightweight structure generally follows the same setup as standard structures, but with smaller exponent and mantissa widths. In addition, there are a number of different rounding/truncation methods that can be used to compensate for possible overflow (when

a positive exponent becomes too large to fit in the exponent field) or underflow (when a

negative exponent becomes too large to fit in the exponent field) of an operation. In [4],

a lightweight floating-point system is developed that uses a 'jamming' rounding

technique and can support variable sizes from 3-bits (1 sign, 1 exp, 1 mantissa) to 32-bits
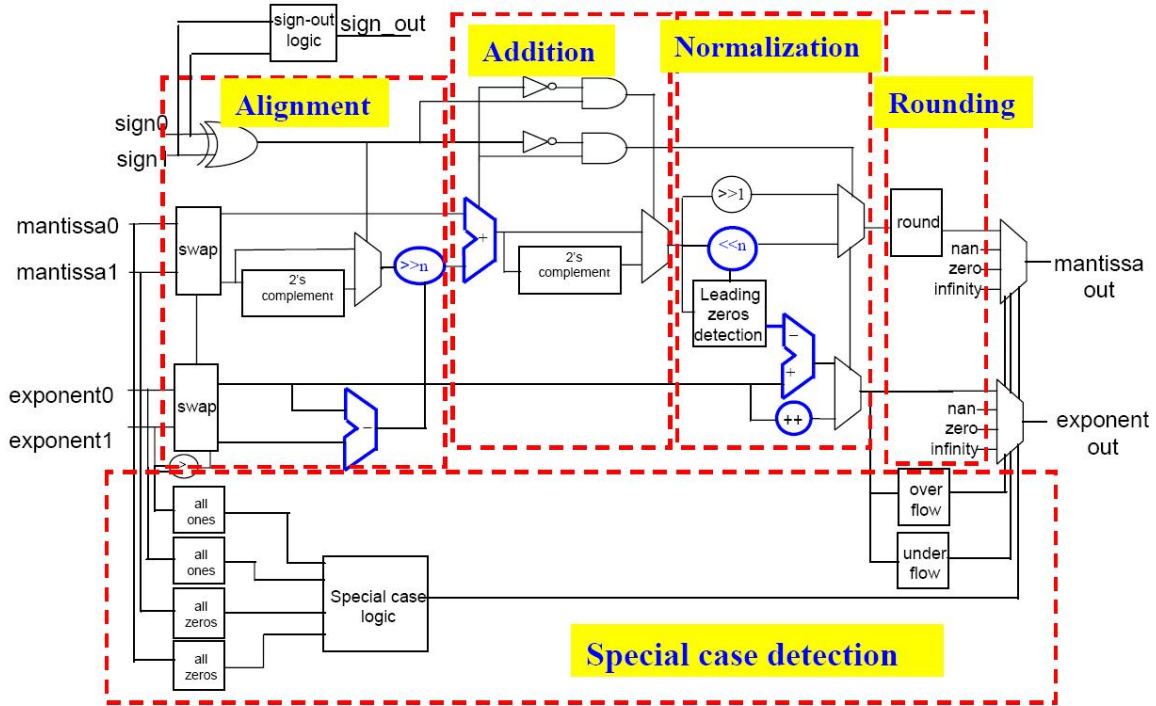
Figure 3.1: Floating-point adder. Blue modules have large area and/or delay.
[4]

(IEEE standard). Chen used a comparison between their 14-bit implementation to that of

the standard 32-bit floating-point, and that of an accuracy comparable 20-bit fixed-point

implementation of adders and multipliers. The results can be seen in Figure 3.3 and

Figure 3.4 respectively. To show a power comparison, they also tested an Inverse

Discrete Cosine Transform (IDCT) in each format of which the results can be seen in

Figure 3.5. By analyzing Chen's results [4], we can see that while a lightweight floating-

point implementation is a great improvement on the standard IEEE format, it still lags the

size and delay aspects of fixed-point in most instances.
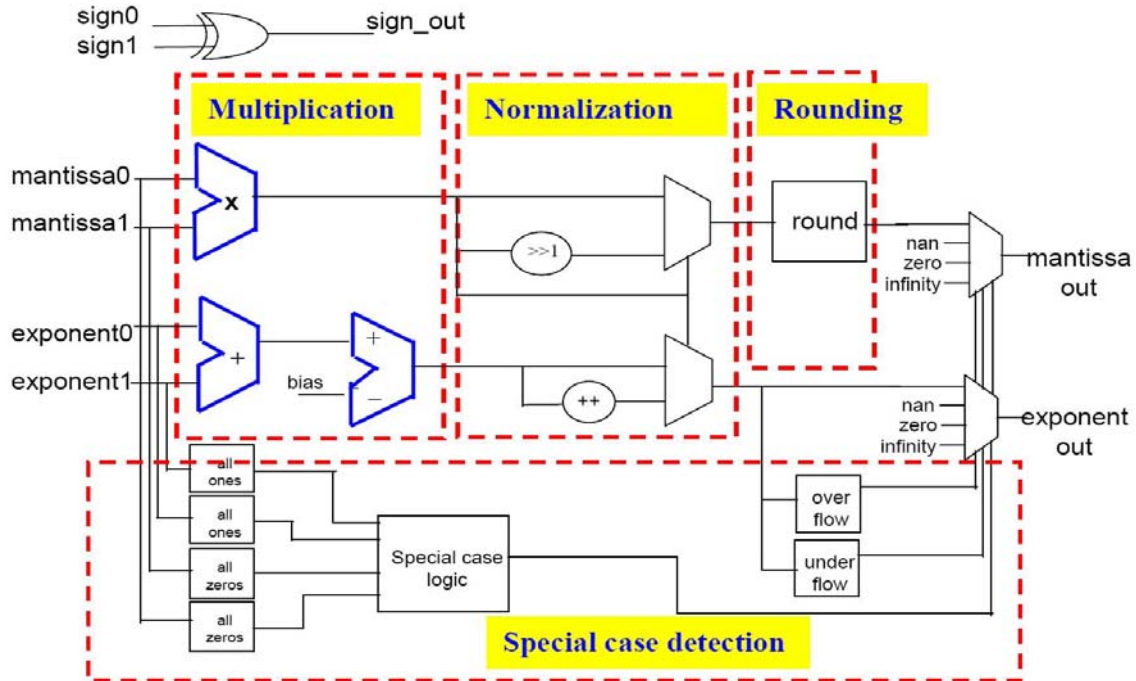
Figure 3.2: Floating-point multiplier. Blue modules have large area and/or delay. [4]

While in the past, fixed-point DSPs held a stronger advantage in size, power, and

speed over floating-point DSPs, and floating-point had an advantage in ease of use, both

of these difference gaps have shrunk significantly. SOC integration and volume has

become more of a factor than the size of the DSP core itself. These issues should no

longer be a deciding factor in which DSP type to use. Instead, the designer must simply

judge the need for range and precision of the project. Fast, high volume sampling rates

such as those used for video applications are better off using a fixed-point approach. On

the other hand, applications that require a wide sampling range and not as much speed, such as audio receivers, will likely be better off using a floating-point approach [5].

32-bit Floating-point

20-bit Fixed-pt

14-bit Floating-pt

	32-bit FP	20-bit FIX	14-bit FP
Area(um^2) - post layout	26634	4866	10096
Delay(ns) - post synthesys	48.95	2.44	25.77
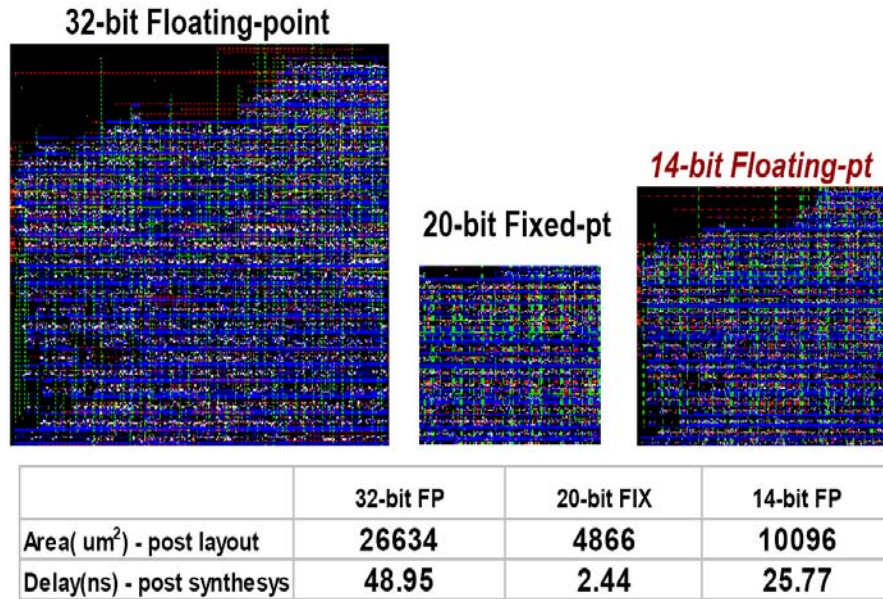
Figure 3.3: Adder design examples. [4]

32-bit Floating-point

20-bit Fixed-point

14-bit Floating-pt

	32-bit FP	20-bit FIX	14-bit FP
Area(um^2) - post layout	60713	40738	8851
Delay(ns) - post synthesis	24.14	22.82	15.89
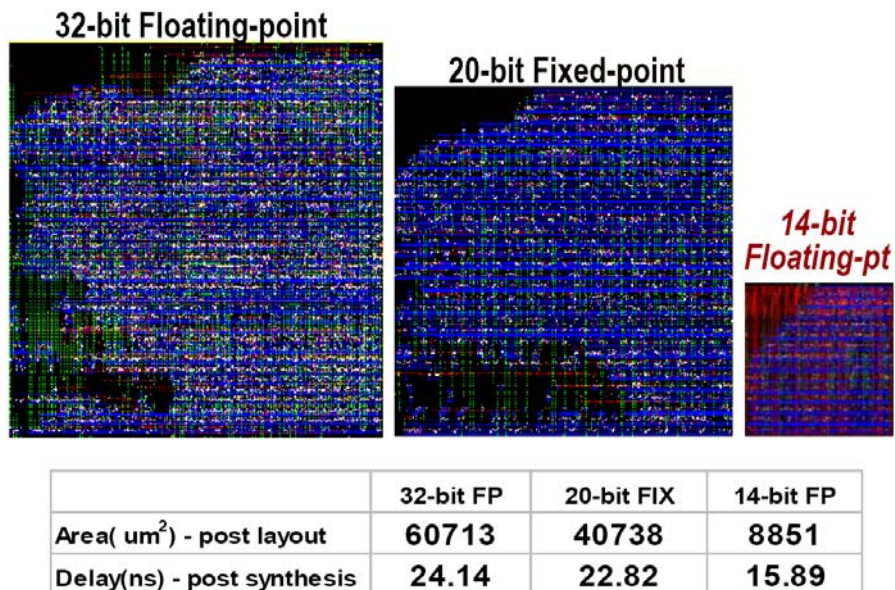
Figure 3.4: Multiplier design examples. [4]
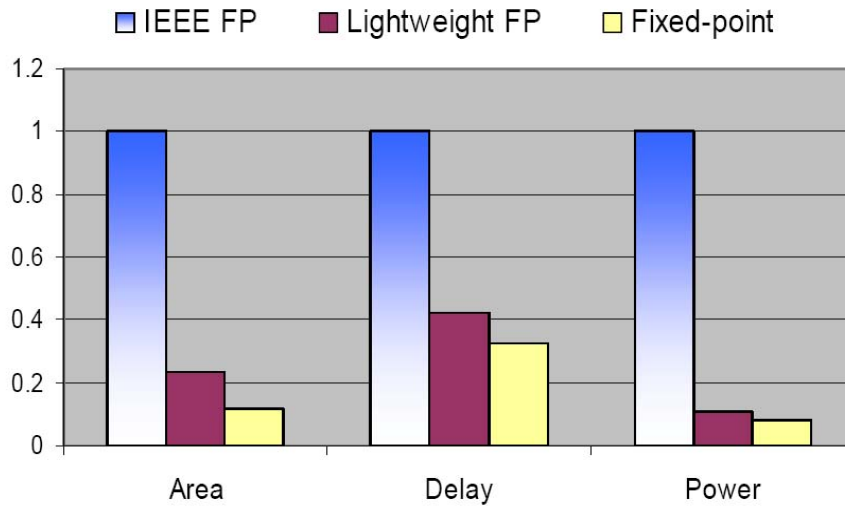
Figure 3.5: Parameter comparisons between versions of IDCTs. [4]

For our purposes, we are looking at a video receiver with small bit-width inputs (likely 2-16 bits), coming at high frequencies. This equates to a lot of small data samples. Since our largest concern is with the ability to process the data at such high speeds, it would be excessive to use a floating-point representation on the input.

Therefore we will stick to a reduced fixed-point representation of the decimal values of the input, output, and filter coefficients of the down-converter. By using a fixed-point binary representation however, we would also need a very significant and unrealistic number of bits to come near a high-precision decimal value. Take for example the decimal: 765.8125. We can see the decimal value has seven significant digits on each side of the decimal point. If we convert the number to binary however, we end up with the value: 1011111101.1101. The decimal equivalent we can see now has

14 significant bits meaning we would need to use a 14-bit signal to represent it to an exact precision.

At high speeds we are more likely to be using signals of somewhere from 2 to 8 bits on an input, maybe even up to 12 bits for a slower device. To use the value above in an 8-bit fixed-point signal, we would need to either round it or truncate it. If using a signed representation (as we will be doing) we may also need to add the first bit to represent the sign (positive = '0' or negative = '1') of the value. The above number using this method would then be truncated to: (0)1011111000.0000, as the last 7 bits would not be able to be analyzed. Appendix 1 contains the Matlab function 'trunk_down.m' that utilizes Matlab's built-in 'fi' functions to convert a long precision value into a binary equivalent based on fixed-point binary parameters [reference Matlab help]. The resulting decimal equivalent of the above, truncated binary value is: 760, which has a 0.76% error from the original decimal value. This may or may not be a significant error depending on the application of the hardware.

Mathematical operations in fixed-point are much more direct than in floating-point. The main difference between the two are that in fixed-point operations, it is often necessary to increase the bit-width between the input and output. Take for example the addition of two 4-bit 2's compliment fixed-point values below:

$$10.10 = -1.5$$
$$+10.01 = -1.75$$
$$=100.11 = -3.25$$

We first of all can see how the decimal point stays in the same place throughout the operation. Secondly we can see that while both inputs were only 4-bits wide, the resulting sum must be 5-bits wide to accurately represent the value. If the Most Significant Bit (MSB) were not included, the result would be 0.75 instead, which is highly inaccurate. This same concept also applies to fixed-point multiplication, but on a greater level. Let us use different values to demonstrate a multiplication:

$$
\begin{array}{r}
01.00 = 1.00 \\
*01.11 = 1.75 \\
\hline
01\ 00 \\
010\ 0 \\
0100 \\
0000 \\
\hline
001.1100 = 1.75
\end{array}
$$

We can see above that multiplication both increases the number of bits needed to represent the number, and also causes a shift in the decimal place. The decimal shift is not normally compensated for during the fixed-point operations, but is considered in the back-end processing of the output. The multiplication of 2's compliment numbers brings the added complications of the sign of the result [1]. The general rule of output bit-widths of signed fixed-point multiplications is:

$$Bitwidth_{out} = Bitwidth_{in1} + Bitwidth_{in2} \qquad (3.2)$$

There have been many methodologies devised for both addition and multiplication of signed fixed-point numbers. Adders such as the 'Ripple Carry Adder'

74

and 'Carry Look-Ahead' adder are well documented for use in VLSI systems. There are also a number of digital multiplier techniques. One such scheme is used by [6] to optimize a serial FIR filter. Figure 3.6 gives an example of the scheme for two 4-bit numbers in which the sign information of the result is contained in the partial products X_i*Y_3 and X_3*Y_i.

				X_3	X_2	X_1	X_0
			$\times$	Y_3	Y_2	Y_1	Y_0
			1	$\overline{X_0Y_3}$	X_0Y_2	X_0Y_1	X_0Y_0
			$\overline{X_1Y_3}$	X_1Y_2	X_1Y_1	X_1Y_0	
		$\overline{X_2Y_3}$	X_2Y_2	X_2Y_1	X_2Y_0		
+1	X_3Y_3	$\overline{X_3Y_2}$	$\overline{X_3Y_1}$	$\overline{X_3Y_0}$			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Figure 3.6: Four-bit multiplication example [6].

The accuracy issue of using a lesser number of bits to represent a signal can be seen clearly in Figure 3.7 where the same input signal is represented with different bit-width values.

The question then is what is an acceptable error for a specific application, and what size bit-widths are required to preserve that error. For this research, Tsui's SSS will be used in which three parameters (input values, filter coefficient values, and output values) must be translated to binary and truncated for hardware specifications. Therefore, by using the truncation code in Appendix A in which we utilize the 'fi' function in Matlab [7], we will test and classify the errors in data (I/Q imbalances) that are caused by such truncation errors.
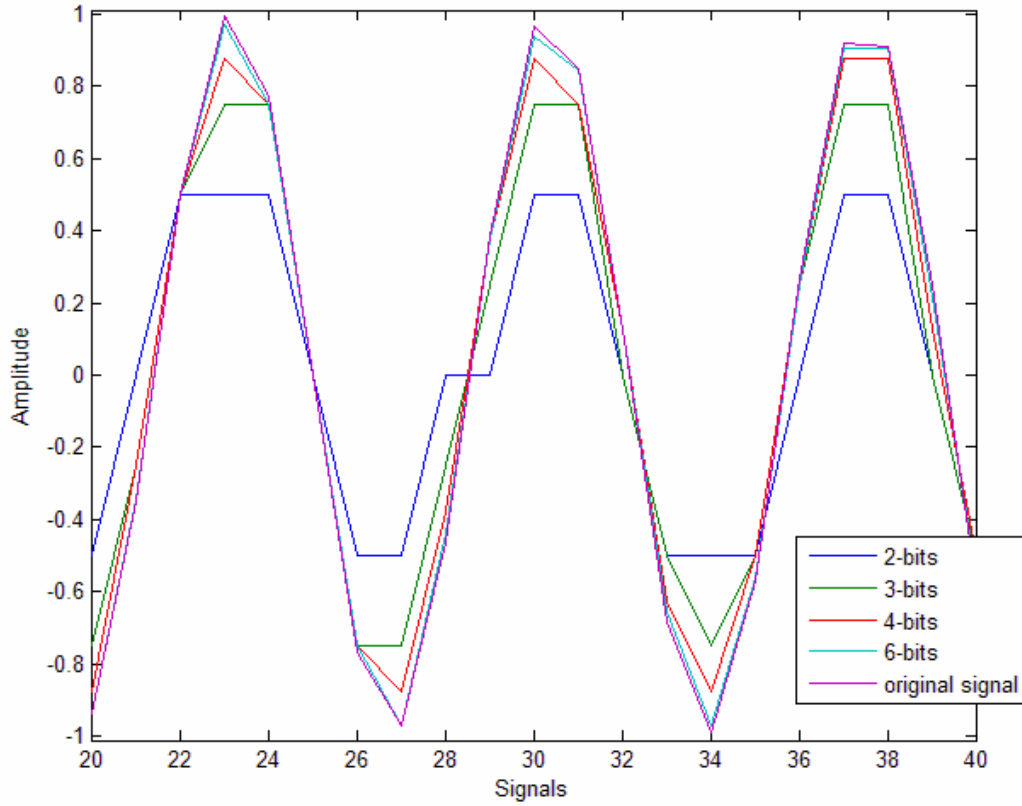
Figure 3.7: Waveforms sampled at different signed, truncated bit-width accuracies.

3.2 Modeling I/Q Data and Imbalance

Before we can begin designing a hardware version of the I/Q SSS, we must verify

its operation and usefulness for accurate signal sampling. To accomplish this we will

explore the methods to be used in characterizing the I/Q SSS, in specific, the

programming efforts made to characterize the data with Matlab will be explained. The

basic SSS and original Matlab strategy is presented in Chapter 2, and in detail in [8, Chap

8]. In order to characterize and model the behavior of the I and Q data channels, some

changes and many additions to the original code from Tsui were made. The code used to obtain the data and analysis in this section is presented in Appendix B.

To keep consistent with the base case studied, we will assume that Tsui used the input frequency of 14 MHz and sampling frequency of 100 MHz because this combination illustrates the down-conversion of the input signal well. We will keep with the combination before moving on to more application specific examples. In actuality, any combination of input and sampling frequencies can be used with this sampling scheme as long as the input frequency is kept to less than half the sampling frequency. If the input is faster than this, harmonic effects will become significant. Through analysis, it has also been found that using an input frequency that is exactly one fourth of the sampling frequency will create unreadable results. This is due to the sampling being split into fourths itself in the scheme.

The first manipulation of the sampled data in the SSS is to send it through a low-pass filter to filter out the upper portion of the mixed (sampled) signal. In the base case, we use a 32-tap Kaiser window for our filter coefficients. The Kaiser window is shown in Figure 3.8 using Matlab's window visualization tool [9]. Any number of filter types could be used, but the Kaiser is the most versatile, so we will use it for our characterizations. The corresponding filter coefficients used with a FIR, gives the plots shown in Figure 3.9.

Once the filter is created and split, it is convolved with the split input samples, creating the I and Q channels. The 14 MHz signal sampled at 100 MHz, and it's resulting I and Q channels (which are at about 11 MHz) are shown in Figure 3.10. The plot shows the results of sampling the input signal 256 times. Two signals, spaced one

77
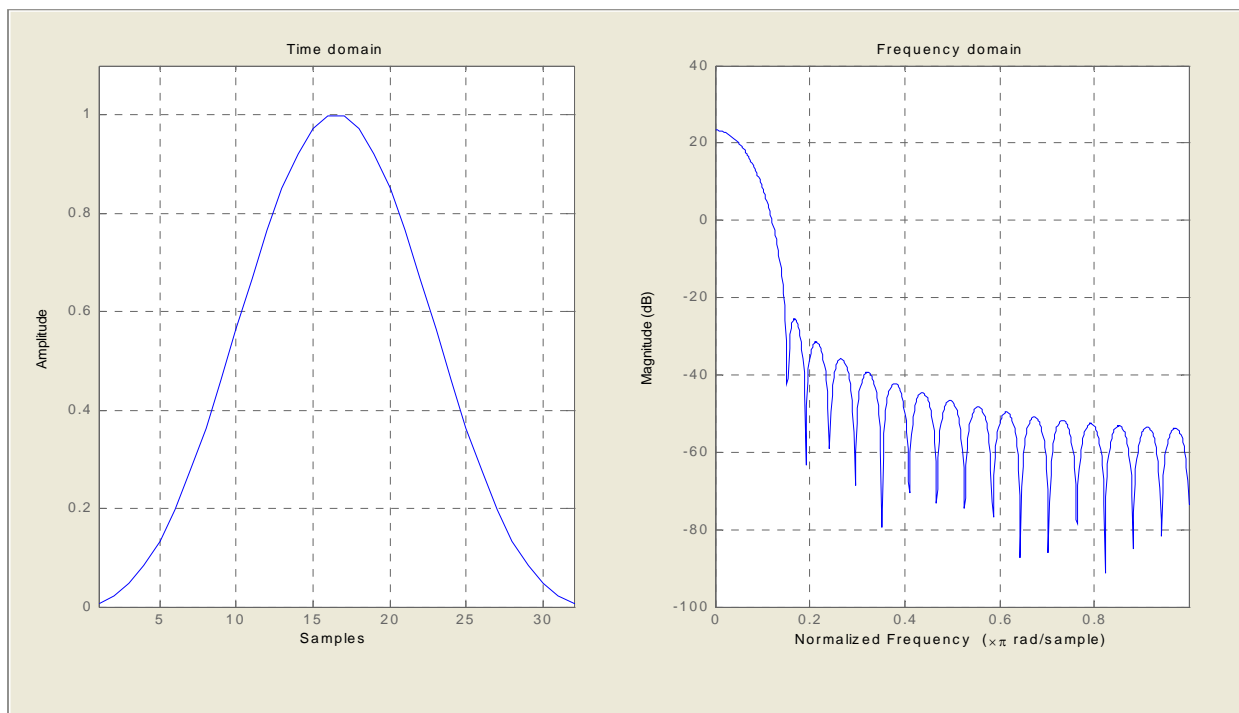
Figure 3.8: 32-tap Kaiser window used for filter coefficients.
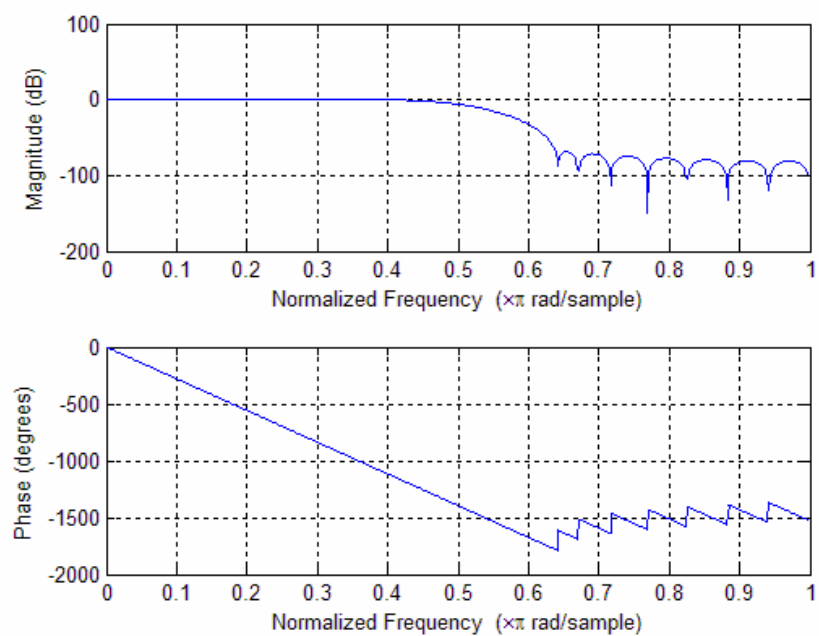
Figure 3.9: FIR plots using a 32-tap Kaiser window.

fourth of a period apart from each other, demonstrate the data channel split into I and Q.

This figure shows the two signals in the time domain.
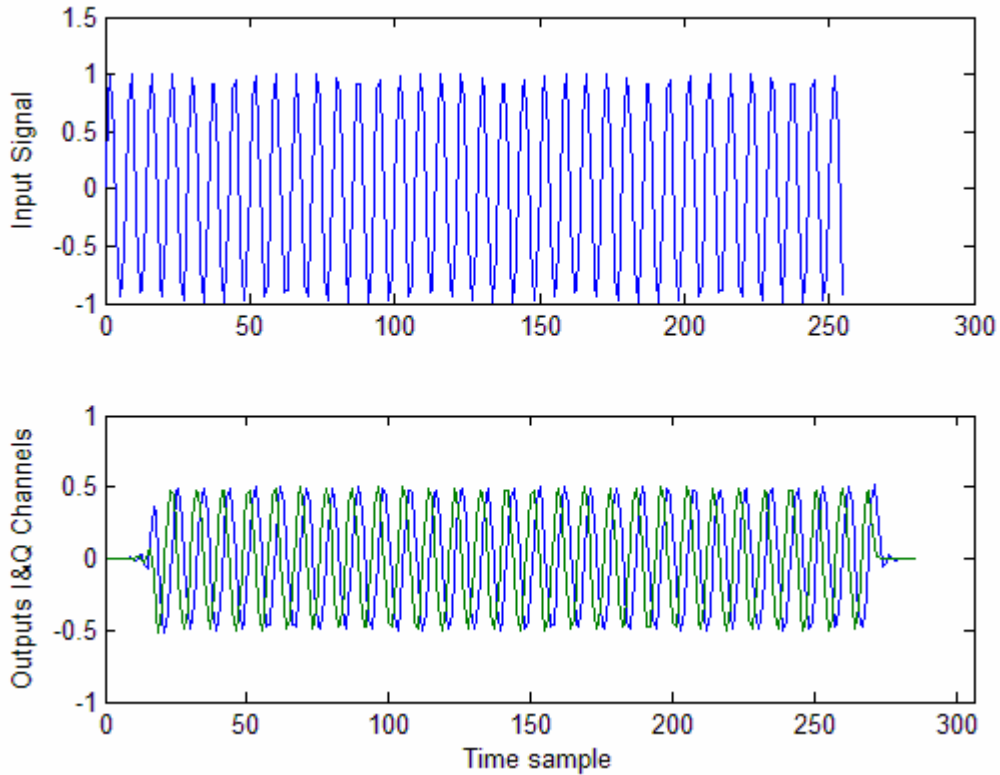
Figure 3.10: 14MHz signal sampled at 100MHz with SSS.

To uniquely described the SSS process, refer to Table 3.2 below. The table lists the parameters needed to successfully define an I/Q downconversion in the SSS.

In order to analyze a signal further into a useful form, we need to produce a frequency representation of them by using a FFT [10]. By keeping the FFT the same length as the number of samples taken we get an equal number of frequency samples. For the case of the I and Q channels, we have more samples than we do for the original signal. This occurs due to the filter convolutions. If a 32-tap filter is used, then we will

79

Table 3.2: SSS parameter list.

Parameter	Description
Input Frequency (or Frequency Range)	Frequency of the incoming signal/s
Sampling Frequency (or Frequency Range)	Frequency of the point sampling of the incoming signal/s
Filter Length	# of coefficients used in the digital filters
Input Bit-Width	# of bits used to represent input values
Coefficient Bit-Width	# of bits used to represent coefficient values
Output Bit-Width	# of bits used to represent output values
Maximum Amplitude Imbalance	Allowable amplitude imbalance between I and Q data channels
Maximum Phase Imbalance	Allowable phase imbalance between I and Q data channels

have 32 more samples in the resulting signal. For the most accurate frequency domain representation, the first and last 16 samples will be disregarded from the I and Q channels, as they simply represent the filter delay and runoff. Therefore, we will end up with three frequency signals of equal number of samples: the original input signal, the down-converted I channel, and the down-converted Q channel. Again, each sample will have a real and an imaginary component. The output of the 14 MHz signal at 100 MHz, sampled 1024 times can be seen in Figure 3.11. Notice the location of the spikes of the I and Q plots are located at the same general locations. When we convert the real and imaginary components of these samples into polar measures, we can get an amplitude and phase plot like the one shown in Figure 3.12. Again, the spurs represent the majority of the signal energy and therefore the points where the spurs occur can be used to determine the amplitude and phase of each signal.
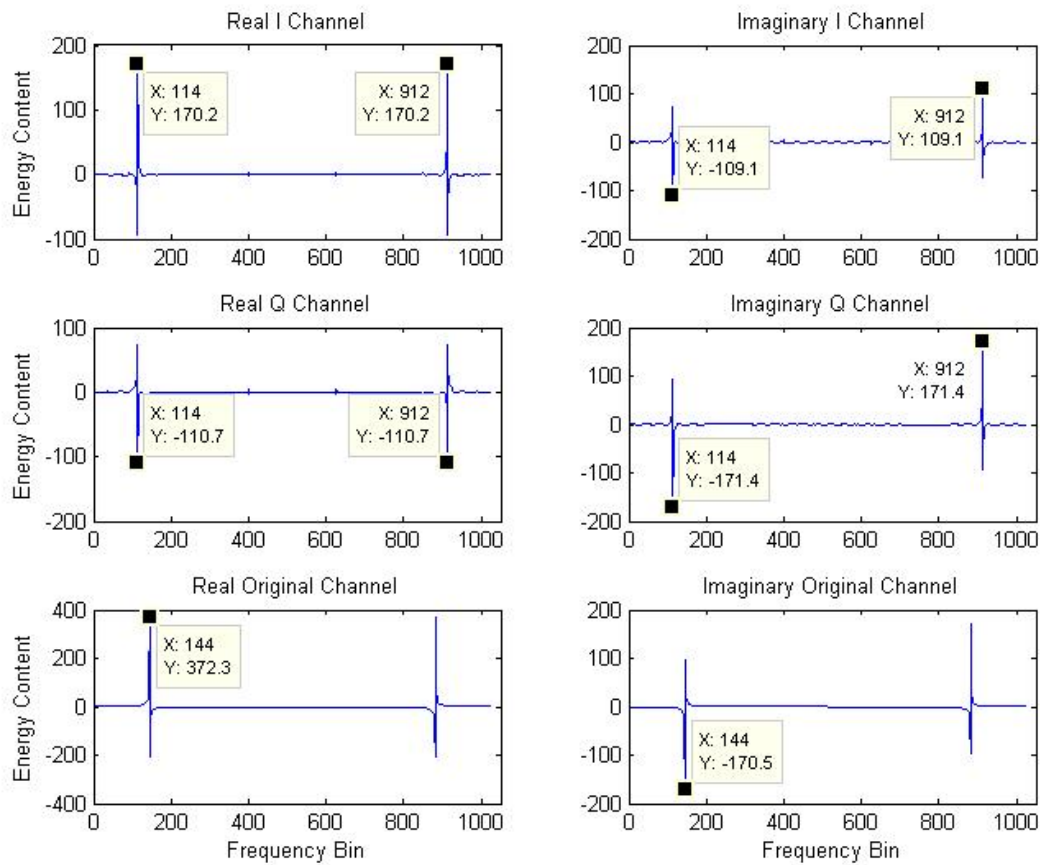
Figure 3.11: FFT plots of 14 MHz signal sampled at 100 MHz, 1024 times with SSS.

From Section 2.7, we know that if the sampling was perfect and ideal, that the I and Q samples will have equal amplitudes and will be exactly $90°$ out of phase. Looking at the location of the spurs in Figure 3.12 and their corresponding frequency values, and looking at the comparison of their values in Figure 3.13, it is clear that the I and Q channels have imbalances in both their expected amplitudes (0.02%) and phases (0.128%). If these imbalances become too significant, the signal may become unusable.
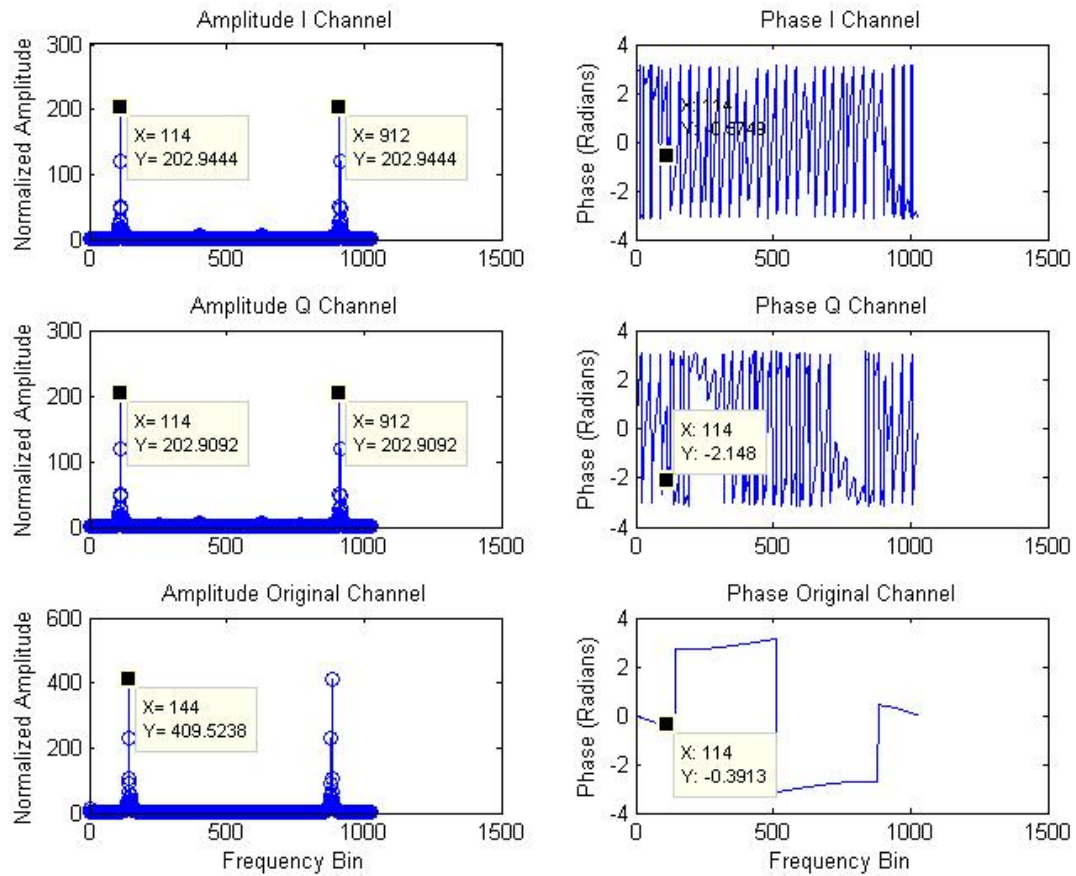
Figure 3.12: Amplitude and phase plots of 14 MHz signal sampled at 100 MHz, 1024 times with SSS.

One final significant manipulation of the FFT signals can help us to verify the frequency of the input and output signals. By taking the power content plot of the signal FFT outputs, we will get a spike at the location of the signal's frequency. If there are two frequencies in the signal, there should be two spikes on the plot. By comparing the power plot of the original signal to that of the I and Q channels, the measure of the down-conversion can be identified. In Figure 3.14, the original signal (14 MHz) has its spike at

82

13.96 MHz and the I and Q channels have their spikes at 11.04 MHz, resulting in a down-conversion of about 3 MHz.
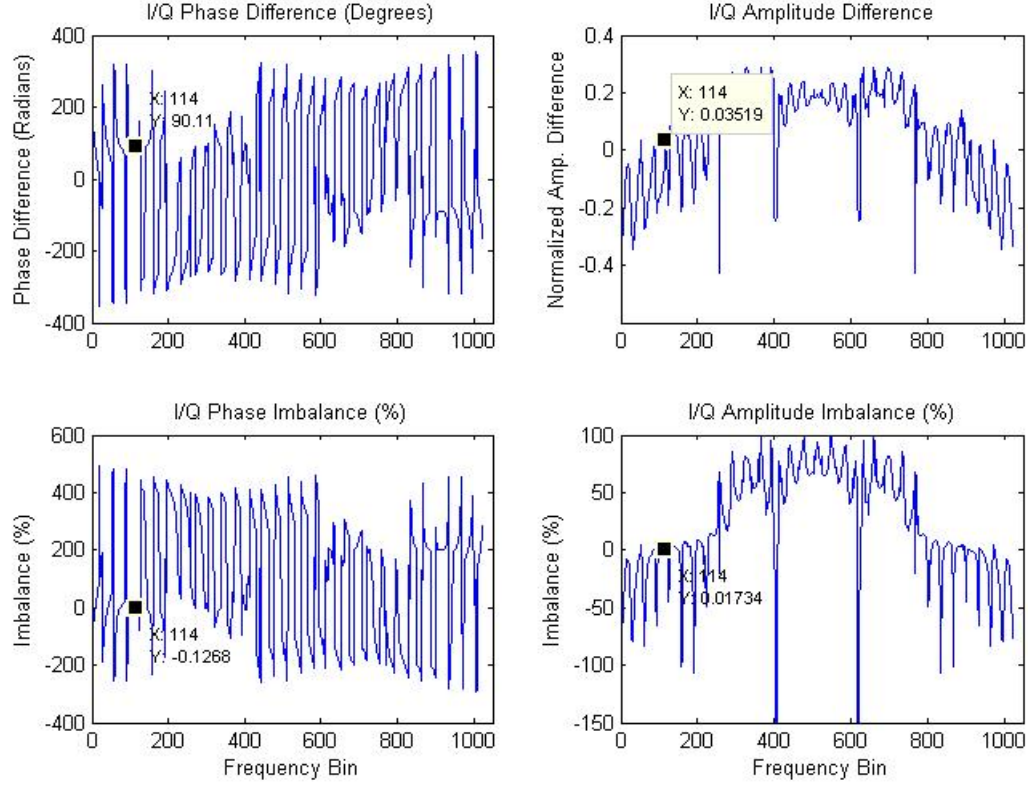
Figure 3.13: Amplitude and phase difference/imbalance plots of 14MHz signal sampled at 100MHz, 1024 times with SSS.

The down-conversion can be verified by subtracting the sampling scheme's resulting sampling frequency (f_s/4), from that of the input frequency. In the case of Figure 3.14, we take:

$$13.96MHz - \frac{100MHz}{4} = 13.96 - 25 = -11.04MHz.$$
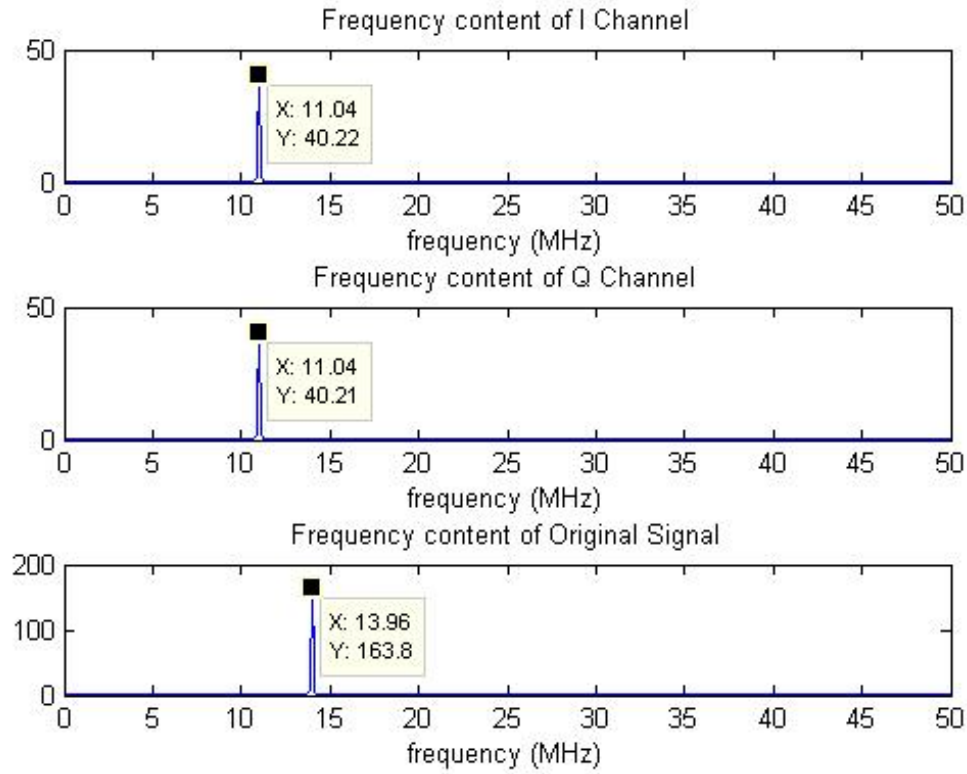
Figure 3.14: Frequency downconversion of 14 MHz signal sampled at 100 MHz, 1024 times with SSS.

Since the frequency domain is symmetrical we can take the absolute value of the result. This means that we will always have a frequency spike in the positive frequency domain.

3.3 Variable Effects on I/Q Imbalances

An all hardware implementation of the SSS is required for use as an ASIC or FPGA. As a result, we will look at five hardware dependent variables that may affect the

imbalance of the I and Q channel outputs: FIR filter length, sample size taken, input bit-width, coefficient bit-width, and output channel bit-widths. Each will be explored and characterized first before combining their effects.

The default filter length in Tsui's original program is 32 tap, but this number can be changed in exchange to get a slightly shorter or longer output delay. Through Figure 3.15 using the default case, the trend is for the phase imbalance to dip to a minimum between 30 and 40 taps, and for the amplitude imbalance to increase as the taps are increased. We also see that 32 taps gives a good balance between the two factors, therefore using 32 taps would seem to be a suitable setting. In Figure 3.16, a faster implementation of 100 MHz signal sampled at 1 GHz is tested to verify a faster use of the
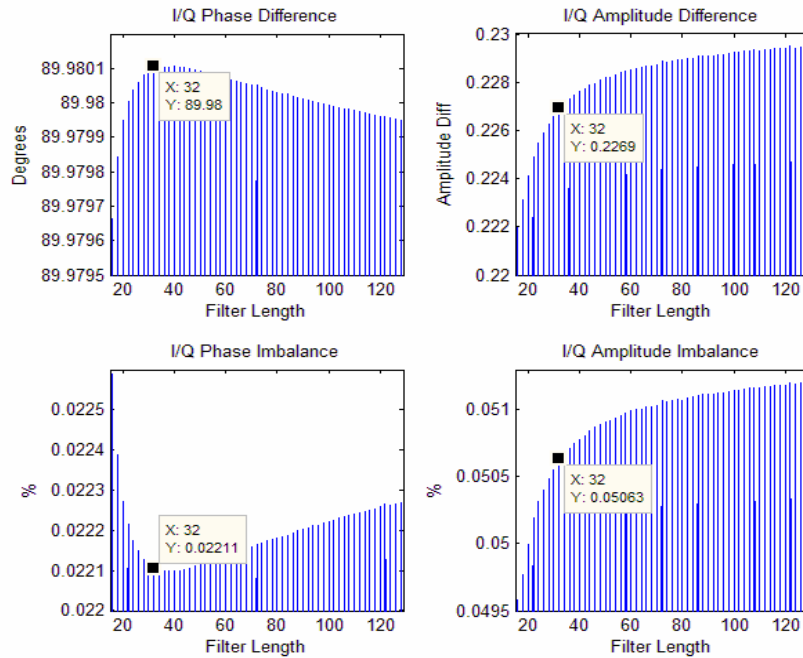
Figure 3.15: Filter length effect on imbalances of 14 MHz signal sampled at 100 MHz, 2048 times with SSS.

sampling scheme. Here we see some similar but not exactly the same trends. Again however, 32 taps seems to be a suitable setting.

The most important aspect to altering the filter length is not the trends themselves, but rather the effect of the trends. In all but the very worst cases, the effect of using any range of filter lengths only seems to affect the overall imbalances by less than 0.005%. As will be seen after analyzing other factors, this is not an overly significant effect, and therefore it will be assume that using the 32-tap filter is a reasonable option for any implementation of the SSS.
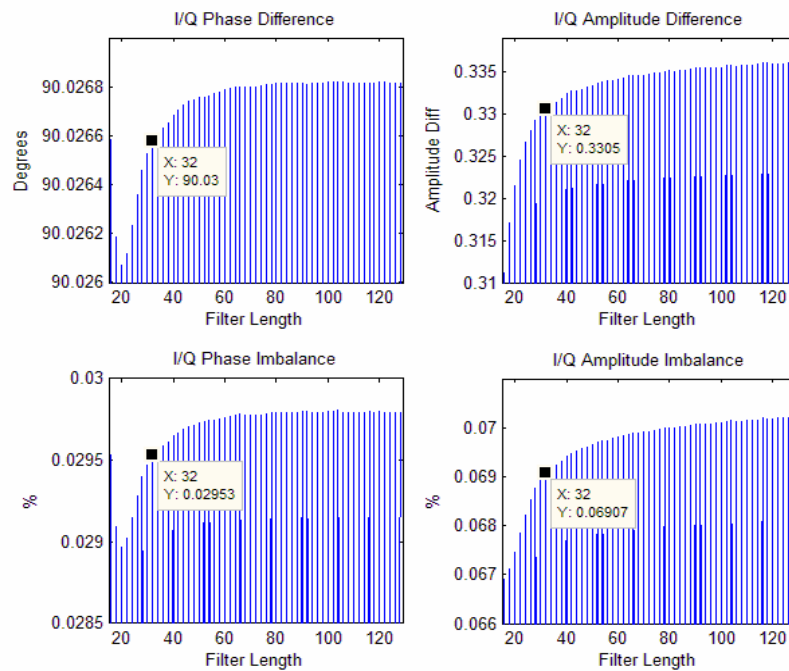
Figure 3.16: Filter length effect on imbalances of 100 MHz signal sampled at 1 GHz, 2048 times with SSS.

The second variable, the sample size of a signal, is the number of samples taken of the input signal at the sampling frequency. A general trend of FFT is to become more

accurate toward the ideal with the more samples that are taken [1]. In a true continuous signal, the number of samples is also continuous and could therefore gravitate towards infinity. To sample however, a certain string of these samples must be extracted and therefore it will not be continuous, but will be a set sampling size.

Looking at the default case in Figure 3.17, the general trend outlined above is verified for imbalances of the I/Q signals as well. We see that a small sampling size such as 128 generates a much greater imbalance (both in amplitude and phase) in the data channels than a larger sampling size. The imbalance trend is almost an inverse exponential as it levels off to an almost linear slope at a sampling size of around 2000. This trend is verified in the case of a 100 MHz signal sampled at 1 GHz as can be seen in
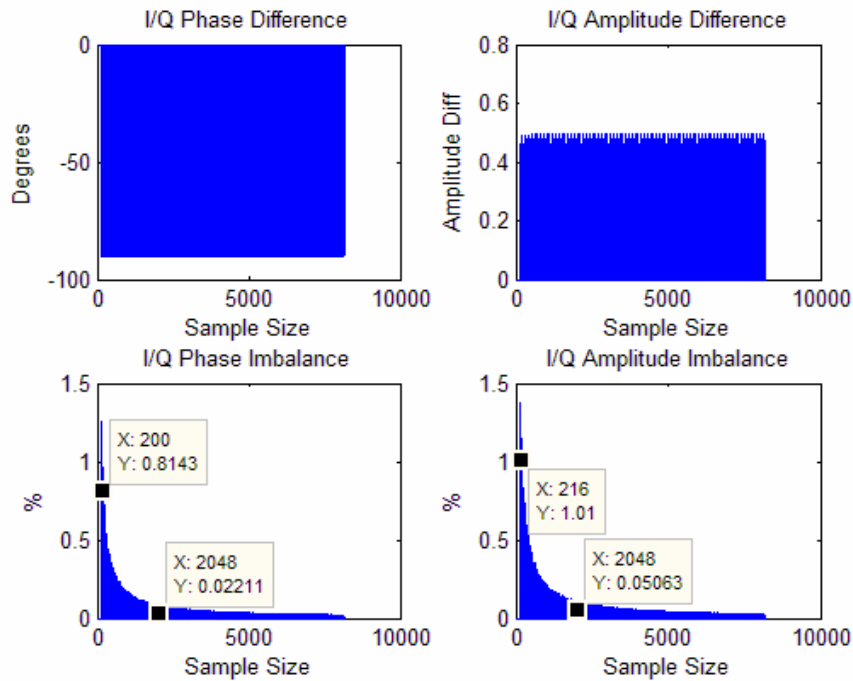
Figure 3.17: Sample size effect on imbalances of 14 MHz signal sampled at 100 MHz with SSS. Uses 32-tap filter.

Figure 3.18. Again the slope is leveling off at around 2000 samples. This will be the case for any legitimate signal/sample frequency combination.

Using the above trend of sampling size effect on channel imbalances, we can use a common digital value such as 2048 as the sampling size for our tests as it will almost always be a suitable value for imbalance evaluation.
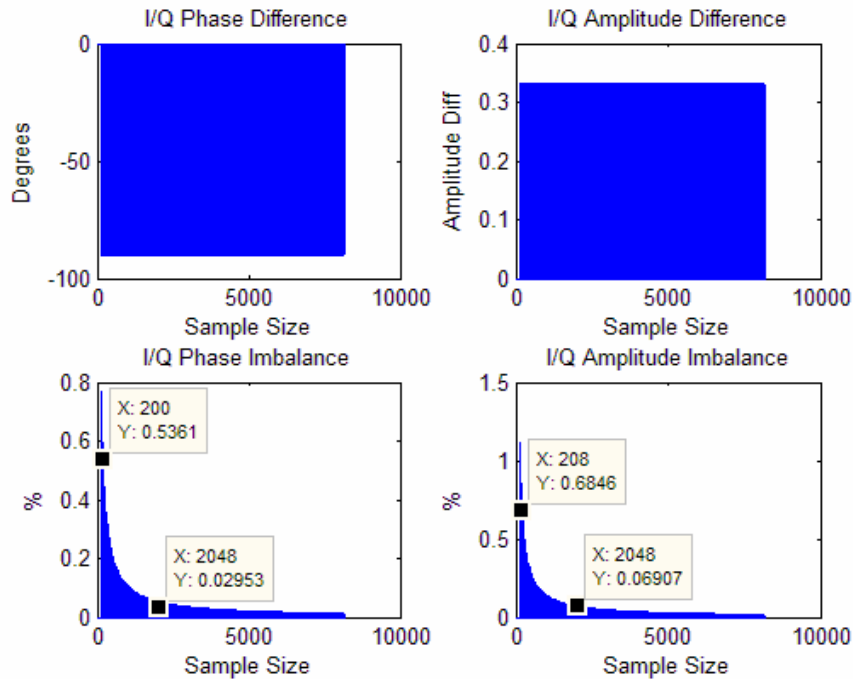
Figure 3.18: Sample size effect on imbalances of 100 MHz signal sampled at 1 GHz with SSS. Uses 32-tap filter.

The next three variables, the bit-widths of the signal values and filter coefficients, are all altered in the same way. They can be represented non-ideally by a specified number of bits. We need to analyze just how reducing these bit-widths will affect the imbalance of the output I and Q channels. Each of these variables will produce a plot of fairly sporadic imbalance values that will level off and eventually become constant as the

number of bits used is increased. In order to create a stable sampling system capable of

sampling a large range of frequencies, we must find a range of bit-widths that will give

consistently low imbalances in the I/Q channels. We especially want to find minimum

bit-widths in which adding extra bits will not improve the imbalances.

The value of the imbalances in the leveled off range of bit-widths will be used for

reference in plots. If a lower number of bits produce a better imbalance in the system

than the reference value, it will be considered acceptable. However, if it produces a

significantly higher imbalance value, it will be considered unacceptable. For our

purposes, we will consider a difference of 0.1% in either the phase or amplitude

imbalance to be significant.

The simplest of the three variables to characterize is the bit-width of the

coefficient values. We can first look at the default case in Figure 3.19, where we see very

stable and near constant values of imbalance which completely level off before 10 bits.

Stable bit-widths can be seen as being three bits or greater. It is clear that the maximum

imbalance does not vary much from the leveled off value and therefore we can assume

that the coefficient bit-width is not significant in this case (much less than 0.1%). We see

that the case of 100 MHz at 1 GHz in Figure 3.20 is a similar plot with the same

conclusions as the default case. Even in some of the worst-case frequency combinations

the maximum imbalance does not differ from the leveled off value by more than 0.1%.

We can therefore conclude that any bit-width used for the filter coefficients, three bits or

greater, may be used without significantly increasing the output imbalance. This will

allow us to adjust this parameter to meet the needs of the system.
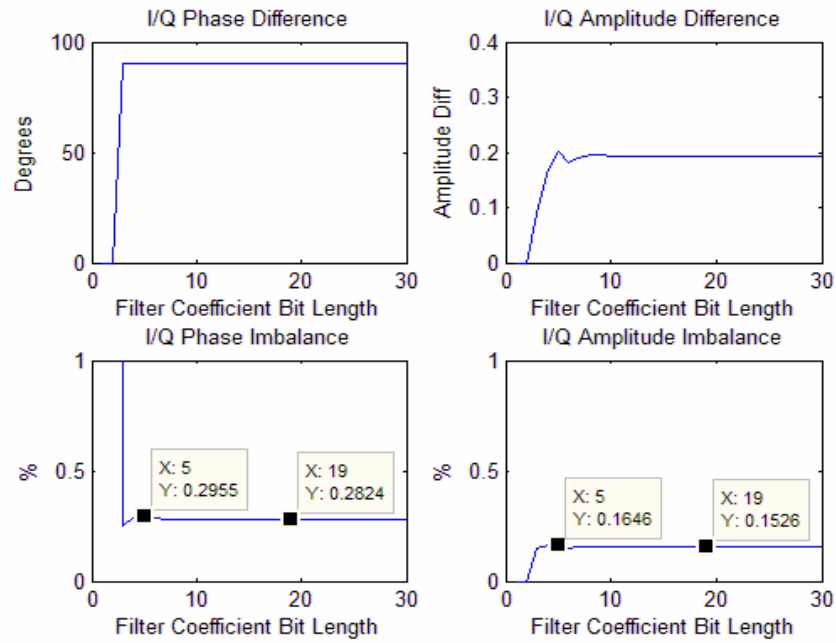
Figure 3.19: Filter coefficient bit-width effect on imbalances of 14 MHz signal sampled at 100 MHz, 2048 times with SSS. Uses 32-tap filter.
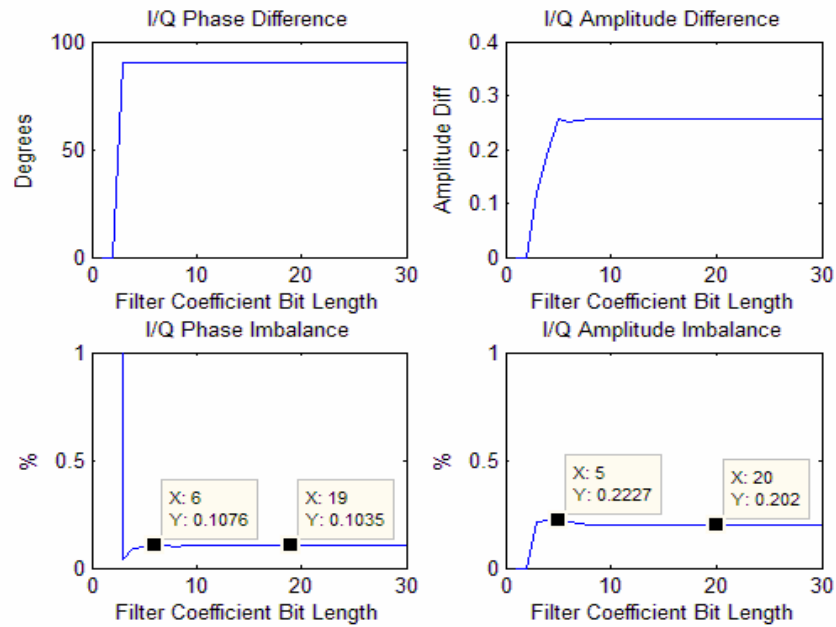
Figure 3.20: Filter coefficient bit-width effect on imbalances of 100 MHz signal sampled at 1 GHz, 2048 times with SSS. Uses 32-tap filter.

The input and output bit-width effects on imbalances are not as consistent as the coefficient bit-widths. Within the lower bit-width values we will see a large variance of imbalances that change sporadically depending on the frequency values used. The default case of 14 MHz sampled at 100 MHz for varying input bit-widths is shown in Figure 3.21. There is a mid-value imbalance peak at 8-bits but we can see that it is not more than 0.1% greater than the leveled-off value and so can be used. We can in fact use any bit value down to 4-bits in this example for phase imbalance. The amplitude
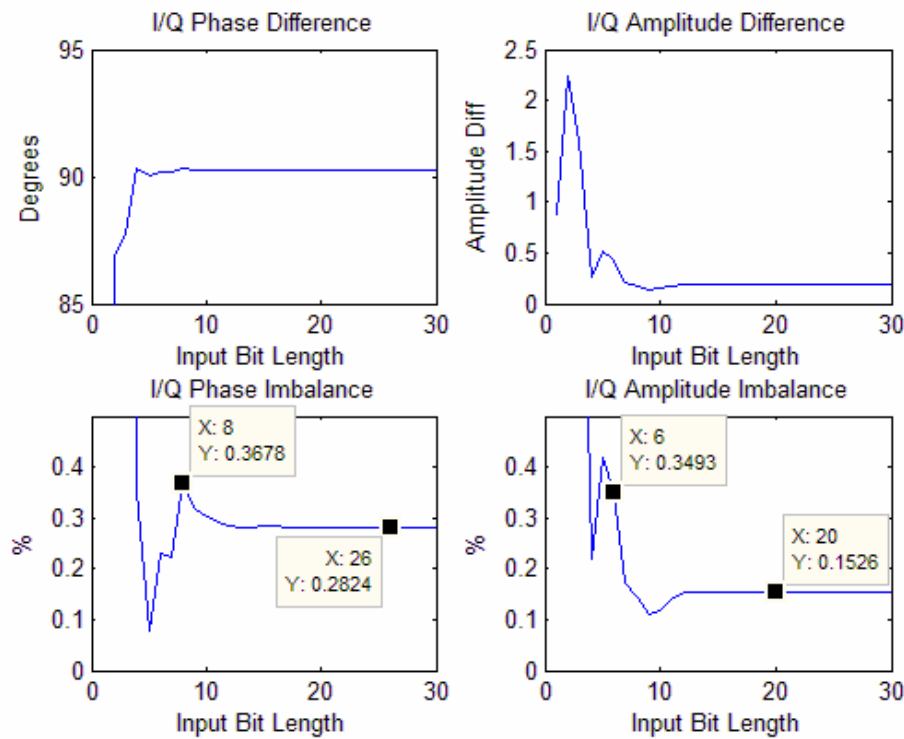
Figure 3.21: Signal input bit-width effect on imbalances of 14 MHz signal sampled at 100 MHz, 2048 times with SSS. Uses 32-tap filter.

imbalance in this example however, shows a mid-value peak at 5-bits which is still above

the acceptable difference at 6-bits. Therefore for the amplitude imbalance in this case,

we need to use a bit-width above six in order to get consistent results. Figure 3.22 shows

an equivalent plot for the secondary case (100 MHz sampled at 1 GHz) in which only

widths above 8-bits can be used for phase imbalance and only above 6-bits for amplitude
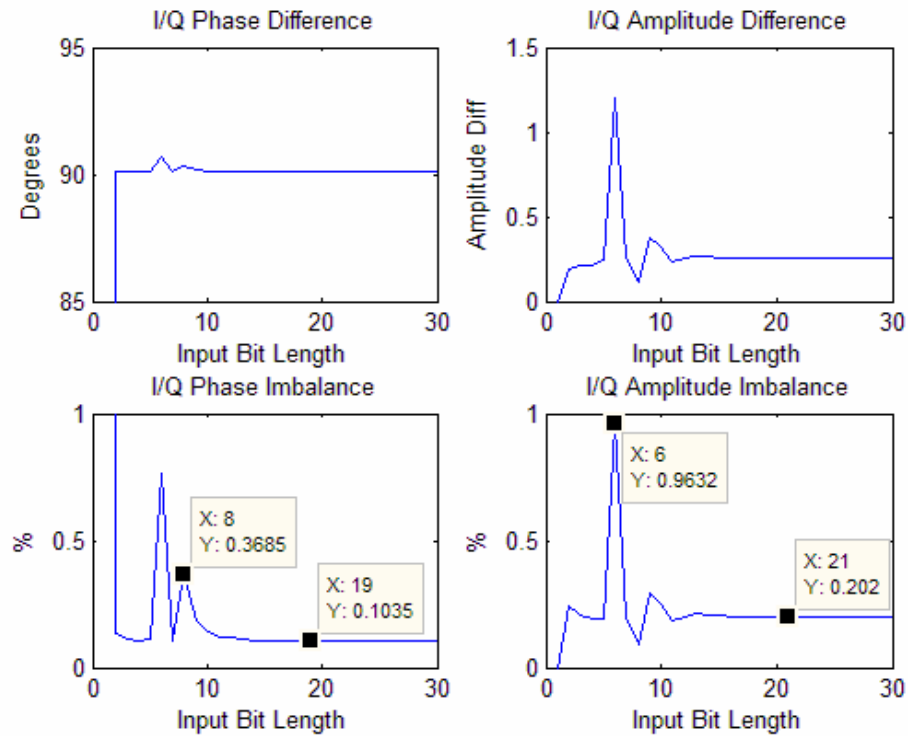
imbalance.

Figure 3.22: Signal input bit-width effect on imbalances of 100 MHz signal sampled at 1 GHz, 2048 times with SSS. Uses 32-tap filter.

Analogous plots to those outlined above are provided in Figure 3.23 and Figure 3.24 for varying output bit-widths. For the default case in Figure 3.23 the mid-value imbalance peaks allow use of bit-widths anywhere above 3-bits for phase imbalance, but only above 7-bits for amplitude imbalance. For the secondary case in Figure 3.24, the use of bit-widths above 8-bits for phase imbalance, and above 7-bits for amplitude imbalance are preferred for consistent results.
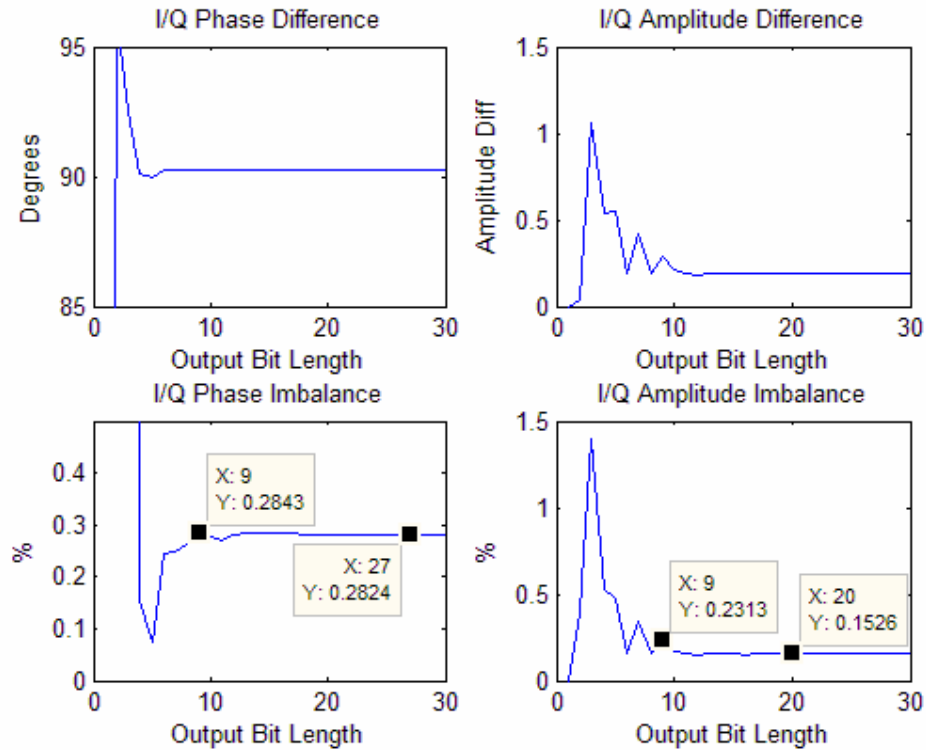
Figure 3.23: Output bit-width effect on imbalances of 14 MHz signal sampled at 100 MHz, 2048 times with SSS. Uses 32-tap filter.
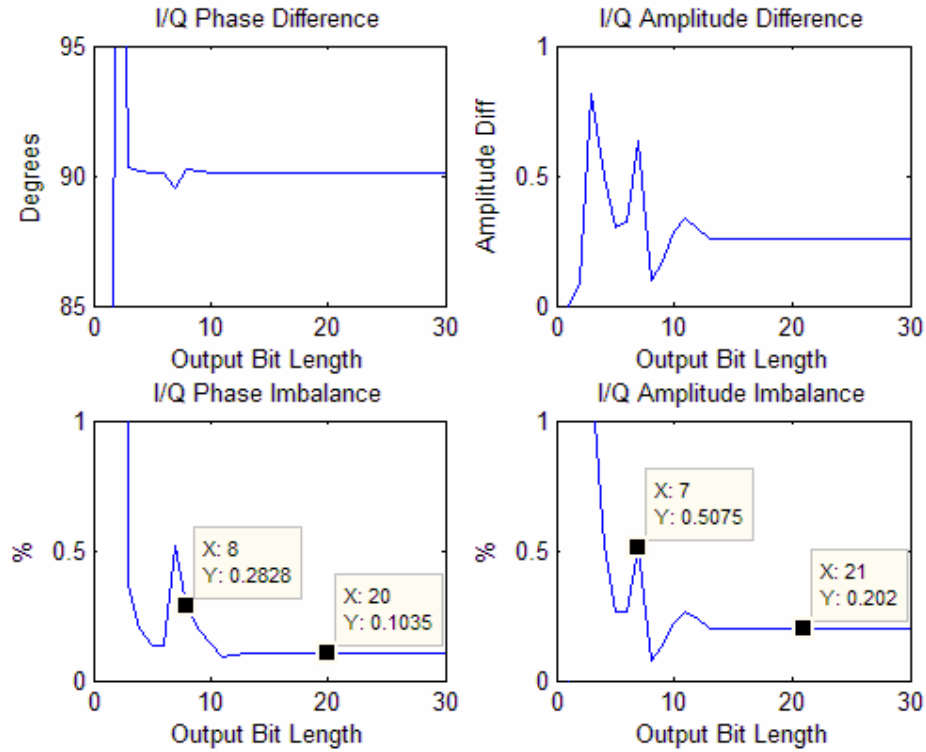
Figure 3.24: Output bit-width effect on imbalances of 100MHz signal sampled at 1Ghz, 2048 times with SSS. Uses 32-tap filter.

3.4 Chapter Summary

In this chapter we took a close look at the elements used to actually sample and characterize the SSS, in an attempt to verify the usefulness of an ASIC or FPGA implementation of it. We looked at potential binary representation methods as well as the FFT operation used to get useful data plots and results. Finally we took an in-depth look at the effect that changing parameters (filter length, sampling size, coefficient bit-width, input bit-width, and output bit-width) has on the I/Q imbalances of the SSS.

3.5 References

[1]   Personal discussion with Col Marty Emmert of the USAF AFRL and Professor at WSU. Dec, 2005.

[2]   Barberis, Marc and Shah, Niraj. *Migrating Signal Processing Applications From Floating-Point to Fixed Point.* Technical Paper, Catalytic Inc. Nov, 2004. Gathered from www.catalyticinc.com on Jan, 2006.

[3]   Patterson, David A. and Hennessy, John L. *Computer Organization And Design.* 3rd Ed. Morgan Kaufmann Publishers, August 2004.

[4]   Chen, Tsuhan. *Lightweight Arithmetic for Mobile Multimedia Devices.* PowerPoint Presentation. January 2006 http://amp.ece.cmu.edu/Publication/Tsuhan/20040309WOCC.pdf.

[5]   Frantz, Gene and Simar, Ray. *Comparing Fixed- and Floating-Point DSPs.* Technical Paper, Texas Instruments. 2004. Gathered from dsp.ti.com on Jan 04.

[6]   Adaos, K. Alexiou, G. Kanopoulos, N. *Development of Reusable Serial FIR Filters With Reprogrammable Coefficients Designed For Serial Dataflow Architectures.* January 2006 http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/7272/19698/00912953.pdf?arnumber=912953.

[7]   MATLAB Fixed-Point Toolbox. "fi function." Acquired from MATLAB help options, 1994-2005 The MathWorks, Inc. 13 January 2006.

[8]   Tsui, James. *Digital Techniques for Wideband Receivers.* Artech House, 2001.

[9]   MATLAB Signal Processing Toolbox. "Kaiser Window." Acquired from MATLAB help options, 1994-2005 The MathWorks, Inc. 10 October 2005.

[10]   MATLAB Function Reference. "Discreet Fourier Transform." Acquired from MATLAB help options, 1994-2005 The MathWorks, Inc. 10 October 2005.

IV. Simulations & Results

4.1 I/Q Results and Imbalance Trends

Since there is no consistent trend in the variation of the imbalances within the lower bit-width of inputs and outputs, we can only guarantee consistently low imbalances at various frequencies by using bit-width values greater than those that fluctuate (10-bits and higher is suitable for all but the most extreme frequency combinations). A table is provided in Appendix D that lists the safe bit-width values of a variety of frequency combinations.

If the frequency combination to be used is of a specific value, individual cases can be analyzed and an optimal combination of input, output, and coefficient bit-widths can be determined. The Matlab code in Appendix C will take parameter limits (maximum bit-widths and imbalances allowed) from the user and output a list of the optimal bit-width combinations for either phase imbalance or amplitude imbalance. These two lists can then be compared to find a combination that will work best for both phase and amplitude integrity.

A very useful trend that was found when comparing frequency combinations was that of matching imbalances when the frequencies were scaled proportionally. Any scaling factor that is applied to both the input frequency and sampling frequency will produce the same imbalances. This trend of proportionality is valid for imbalances caused by both input and output bit-widths, so long as they are kept consistent for each frequency combination. Figure 4.1 shows the imbalances caused by changing the bit-widths of the input of a 14 MHz signal sampled at 100 MHz. In Figure 4.2 the two

frequencies have been scaled by ½, but as we can see, the values match those of Figure

4.1 exactly. The same pattern can be seen in Figure 4.3 and Figure 4.4, which show the

imbalances of changing the output bit-widths. This proportionality can be attributed to

the sampling points being adjusted consistently throughout the sampled signal. In other

words, if the frequencies are scaled proportionally, the sampling frequency will always

sample the input signal at the same point on it's slopes.
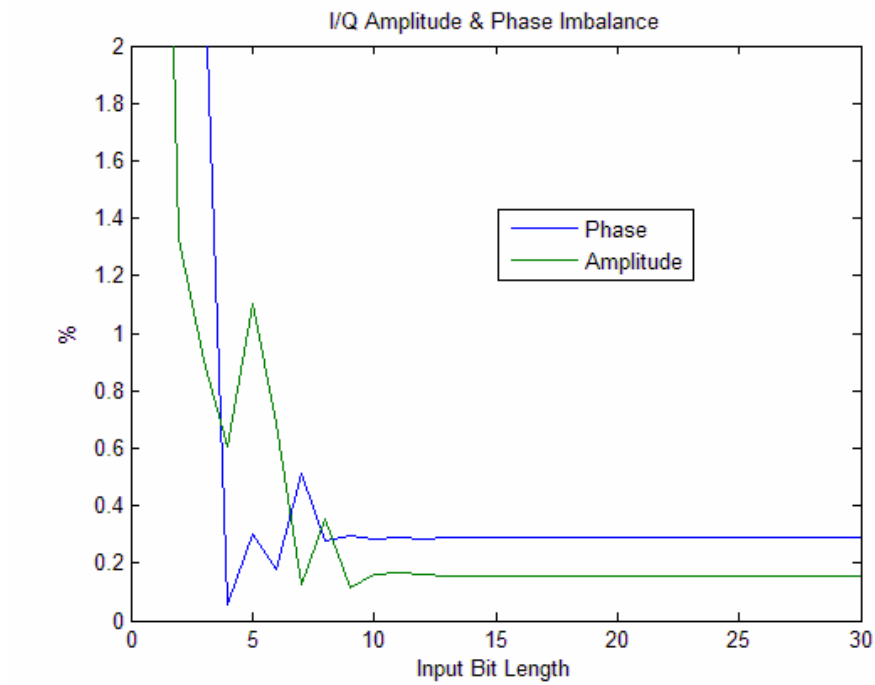
Figure 4.1: Imbalances of input bit-width at $f_i = 14$ MHz, $f_s = 100$ MHz.
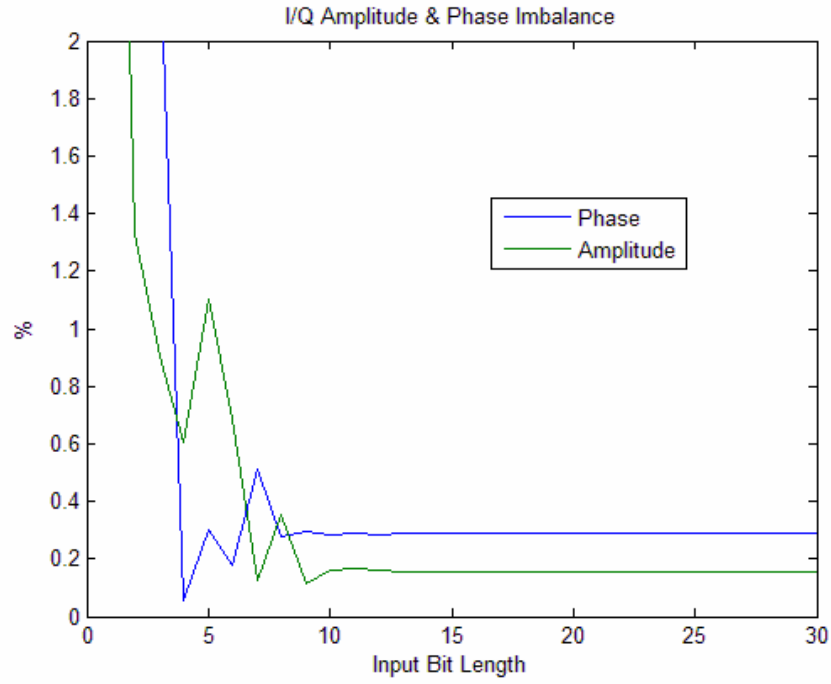
Figure 4.2: Imbalances of input bit-width at $f_i = 7$ MHz, $f_s = 50$ MHz.
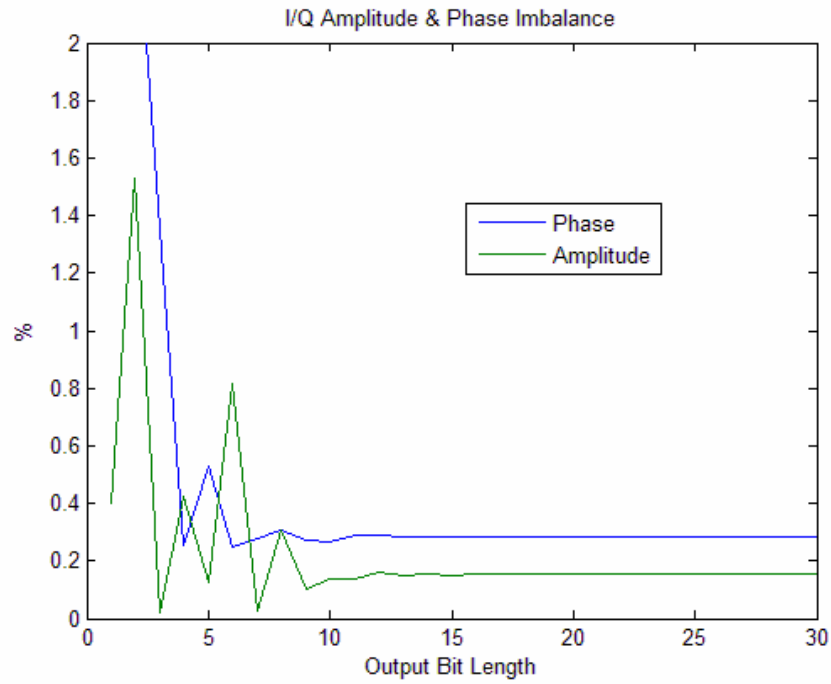
Figure 4.3: Imbalances of output bit-width at $f_i = 14$ MHz, $f_s = 100$ MHz.
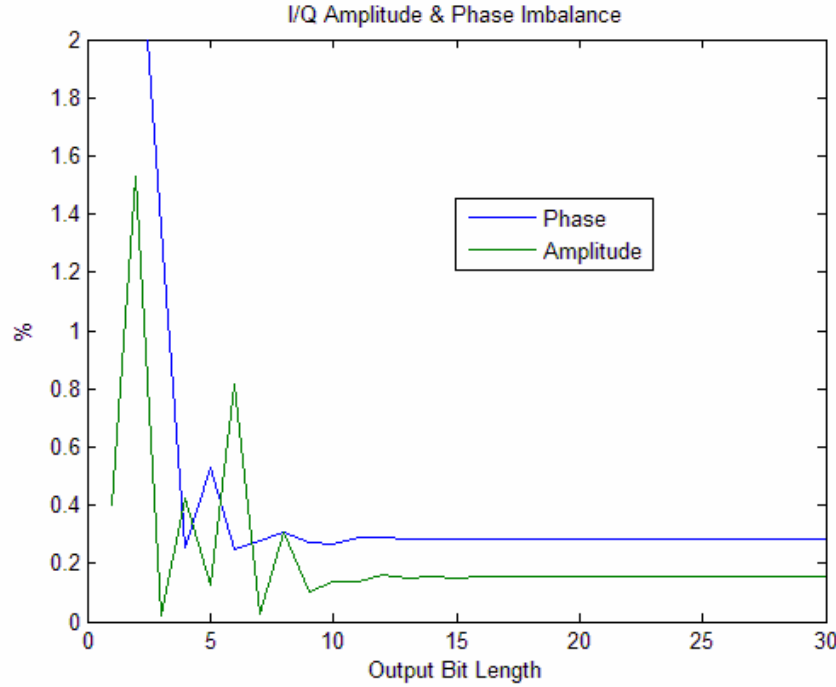
Figure 4.4: Imbalances of output bit-width at $f_i = 7$ MHz, $f_s = 50$ MHz.

The proportionality trend is extremely useful to us in this study as it allows us to test the SSS with lower frequency methods such as an FPGA, before designing it to be implemented for onto a high-speed ASIC. For example, if we were to design a digital downconverter to sample a 1.25 GHz signal at 3 GHz, we could test the design with an FPGA first by sampling a 125 MHz signal at 300 MHz.

4.2 VHDL 'real' validation

The first step to creating an abstract VHDL equivalent representation of the SSS was to make a mathematical equivalent using floating-point (using the 'real' variable type) variables in the VHDL library. The 'real' variable type uses double precision

floating-point math within the computer running the simulations, making the numbers as close to exact values as possible. This however is neither practical, nor synthesizable for a true hardware implementation (due to speed and size requirements), and therefore this can only be used to test the concept.

Conceptually, the SSS can be represented by three different stages: data sample splitting of the input signal, signal filtering of the split data streams, and recombination of the filtered data streams. This process is demonstrated in Figure 4.5 below. All VHDL code was developed from a skeleton example of input data steam control demonstrated by Matlab's Filter Design HDL Coder [1, 2].
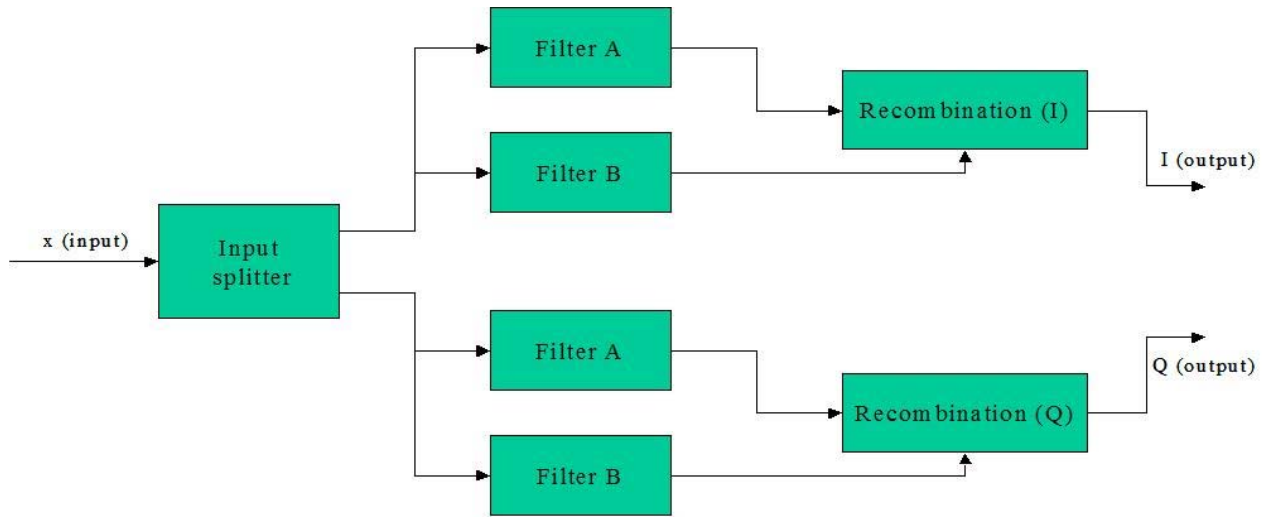
Figure 4.5: Block diagram of SSS modules.

The first stage in the process contains only one module; the input data splitter. The process is very simple and corresponds to Equation (2.17) and Equation (2.18) from Section 2.2. The basis is that the input data stream is split into four separate streams as in Equation (2.17):

$$X1(t) = x(1), x(5), x(9)...$$
$$X2(t) = x(2), x(6), x(10)...$$
$$X3(t) = x(3), x(7), x(11)...$$
$$X4(t) = x(4), x(8), x(12)...$$

(4.1)

$X3$ and $X4$ can be inverted, and the signals $X1$ and $X3$, and $X2$ and $X4$ alternately recombined into:

$$X_r(t) = x(1), -x(3), x(5), -x(7)...$$
$$X_i(t) = x(2), -x(4), x(6), -x(8)...$$

(4.2)

This process is accomplished in Appendix B by using the 'reshape' function. In VHDL however, we use a 'case' statement to track which variable we are currently on. The VHDL implementation of IQgen.vhd, which utilizes 'real' variables, is given in Appendix E.

The second stage of the SSS is the actual filtering stage and this is where most of the speed, size and power of the design will be consumed. As was demonstrated in Section 2.2, we need to filter each of the inputs (*xr* and *xi*) to this section by convolving them with a set of coefficients. Instead of doing this in one filter however, we split the coefficient into two separate filters: 'filtat' and 'filtbt'. To create these filters, the Matlab Filter Design HDL Coder was used to create an exact VHDL representation of the Kaiser window FIR filter used in Appendix B. The code was then adjusted from the 32-tap filter created by Matlab, into two 16-bit filters that utilize a 'Generic' parameter list and use the 'generate' function as described in [3, 4] to cut down on coding length and

101

create a more abstract representation. The VHDL code for IQfir_filtat.vhd and IQfir_filtbt.vhd is given in Appendix F and Appendix G, respectively.

The final stage of the scheme is to recombine the outputs of the four filters into two output data signals: I and Q. This is a simple process of alternating every-other value between 'cnvlra' and 'cnvlrb' for the I channel, and between 'cnvlia' and 'cnvlib' for the Q channel. Of special note is that a delay signal is used in order to make the first Q channel value zero and shift each proceeding value one place over. This module is given in IQoutput.vhd in Appendix H.

The overall design is tied together using a purely structural architecture that represents Figure 4.5 exactly. This VHDL code (IQ_sss_converter.vhd) is given in Appendix I. Testing this code on Symphony EDA Sonata 3.0 with our default signal case of 14MHz signal sampled at 100MHz, we get outputs as shown in Figure 4.6. These signals are exact representations of those in Figure 3.10, and as such, share the same numerical values for each sample point.

With an accurate and working validation of the Matlab sampling scheme process. We next turn our attention towards a purely binary representation that will implement input, output, and signal values of only 'std_logic' or 'std_logic_vector' types, rather than the 'real' type. This is needed in order to make the code synthesizable by industry synthesis tools. The design structure is the same as in Figure 4.5, but the inside coding and structures can be changed within each block element. Much as the 'real' code representations were given in Appendices E through I, the binary equivalents are presented as follows:

Appendix J: IQgen_dig.vhd

Appendix K: IQfir_filtat_dig.vhd (equivalent code used for filtbt)

Appendix L: IQoutput_dig.vhd

Appendix M: tsui_sss_converter_dig.vhd
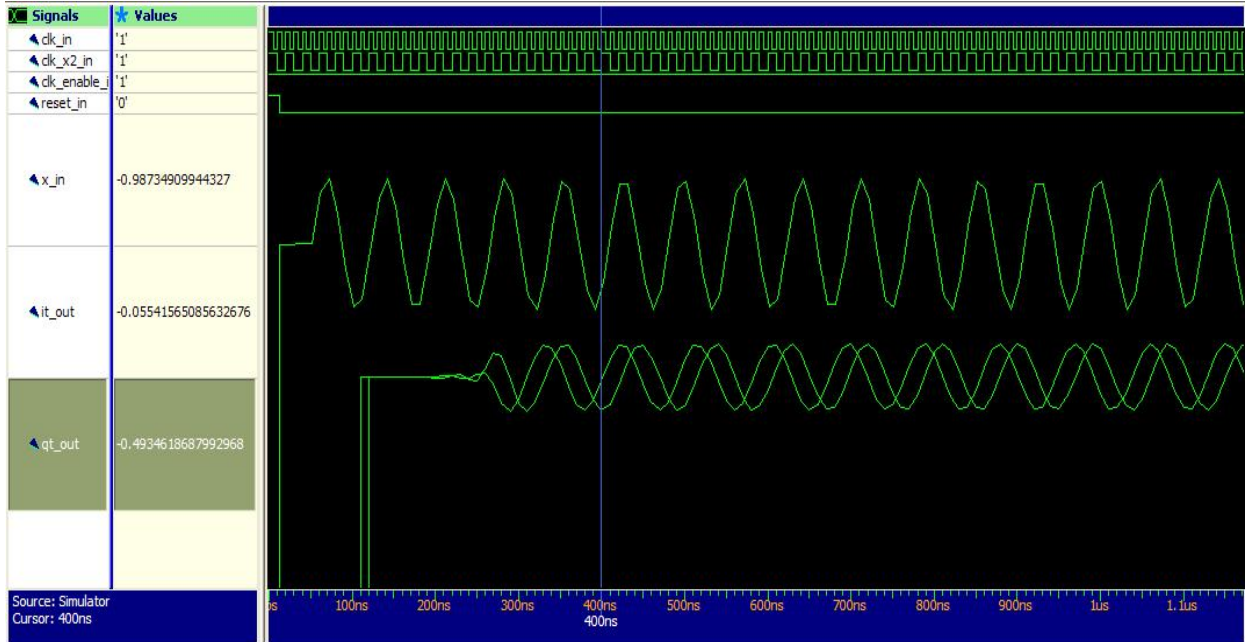
Figure 4.6: Output of IQ_sss_converter.vhd using sample values of 14MHz signal
sampled at 100MHz. I and Q data channels are aligned.

When comparing the digital versions of the code to the 'real' or analog versions,

we can see that all 'real' types have been replaced and initialized appropriately. We also

begin to see a change in such operations as sign conversions in the IQdig files and

especially in arithmetic operations in the IQfir files.

103

4.3 VHDL optimization and results

The digital adjustments made at the end of Section 4.2 take us much closer to an adjustable, synthesizable design, but there are still a number of aspects that need to be dealt with. In the end, we want easily adjustable, fully synthesizable codes that can be optimized for either size, speed, or somewhere in the middle. The most obvious starting spot for such is in the coding of the most complicated block of the design, the filters, and their inner arithmetic operations. The filters are where the great majority of size and speed limitation take place in this design, and therefore are the vital components for optimization. The codes developed in the previous section use a top-level algorithm that could be interpreted many ways depending upon a compiler. This being the case, there is no way to know for sure, how a synthesis tool is going to implement an operation such as a simple addition or multiplication. Chances are, the implementation will not be optimized for either size or speed, but will instead lie somewhere in-between. Therefore, we need to develop specific architectures that will be close to optimal for size or speed so that the user has the option to use either.

The first optimization will compensate for speed. The easiest way to increase the speed of a design is to utilize a design strategy known as 'pipelining.' A pipelined design uses memory registers that can hold a signal for a clock period before moving the value on to the next gate. Such registers will cause a delay from the first input to the first output, but once the first output is accomplished, there can be a new output on every clock cycle. It can be thought of as the information moving through pipes or wired of the circuit, hence the name 'pipelining.'

The only limiting factor to the pipelining speed is in the slowest operating unit, such as an adder for example. Just as the flow of water flow through a pipe can only go as fast as it does through the narrowest part of the pipe, information can only be passed through a circuit at a clock period corresponding to the slowest unit of the circuit. Since a new output in a fully pipelined design can be accomplished in each clock cycle, it can potentially run as fast as the fastest clock period, and is therefore optimized for speed [5, Chap 6]. This strategy does, however, cause the design to be larger than a non-pipelined one because each signal must also have a corresponding register unit to store its value. A block diagram of a common pipelined FIR filter is shown in Figure 4.7. A second block diagram of the pipelined FIR filter that we use in our design (as it is better optimized for speed and has a shorter delay) can be seen in Figure 4.8. Notice that at each 'level' of processing, there is a column of registers that will hold the value of the previous clock cycle's operations. Each level will cause a one-clock cycle delay from the first input to the first output.

To create a pipelined design in VHDL, a register component (module) must be created and added to the digital design of Section 4.2. All entities are combined and put into one VHDL file to create a near-fully pipelined template that is presented in Appendix N. In order to make the design even more pipelined, registers are also added to the multiplier entity. It can be noticed that the registers and adders in the multiplier entity are created using a 'generate' function to reduce coding and make the design more abstract. Generic variables are also used to add abstraction to the entities, and generic maps are utilized in component calls so that only one set of generics at the very top-level
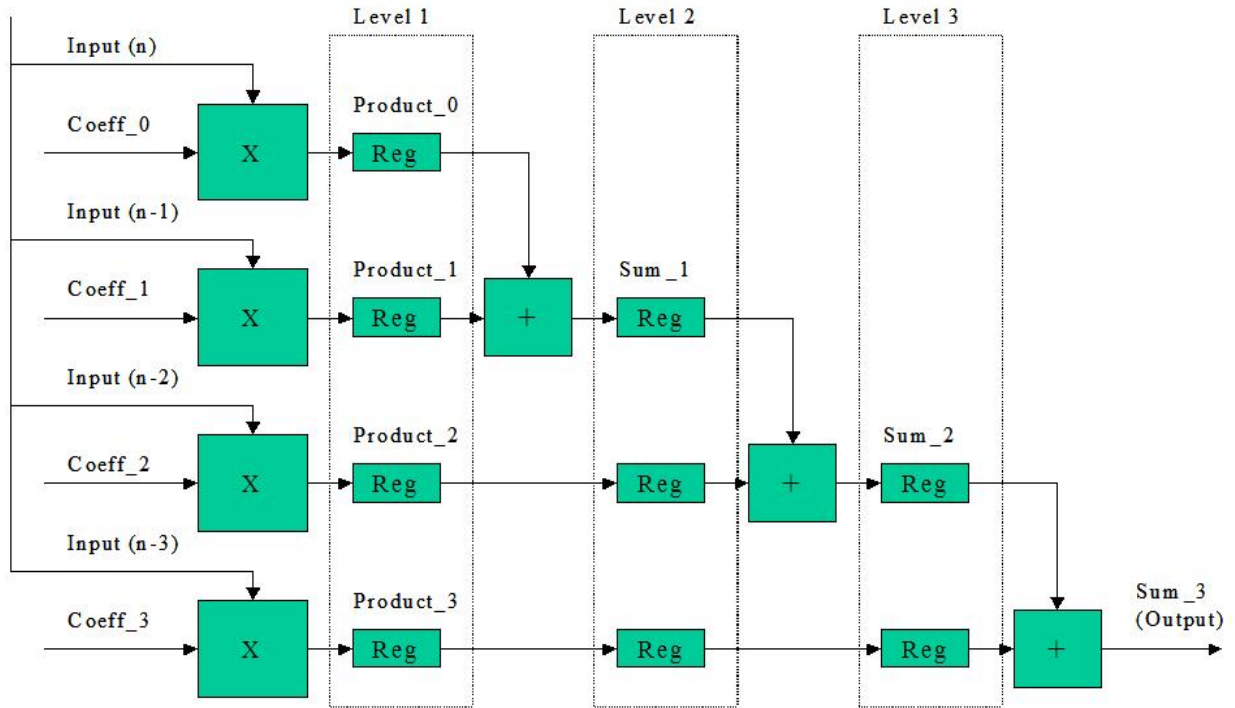
Figure 4.7: Block diagram of a pipelined FIR filter using multipliers, adders, and registers. Number of levels equal number of coefficients minus one.

entity (tsui_sss_converter_struct2_pipeline_dig) must be edited to change a design parameter. Such changes as these make it much easier to create a Perl script that will automatically generate usable VHDL SSS circuits.

A fully sequential design lies on the other end of the size vs. speed tradeoff. In a sequential design, state machines such as shifters and MUXs are used to keep track of which stage of the operation is being done, and to reuse the circuit components more than once in each operation. For example, if we were to do the same process as in Figure 4.7 using a sequential design, we could eliminate all of the operation elements except for one multiplier and one adder. Instead of using three separate adders for the three required additions, we would instead use only one adder, but use it three times. Instead of using
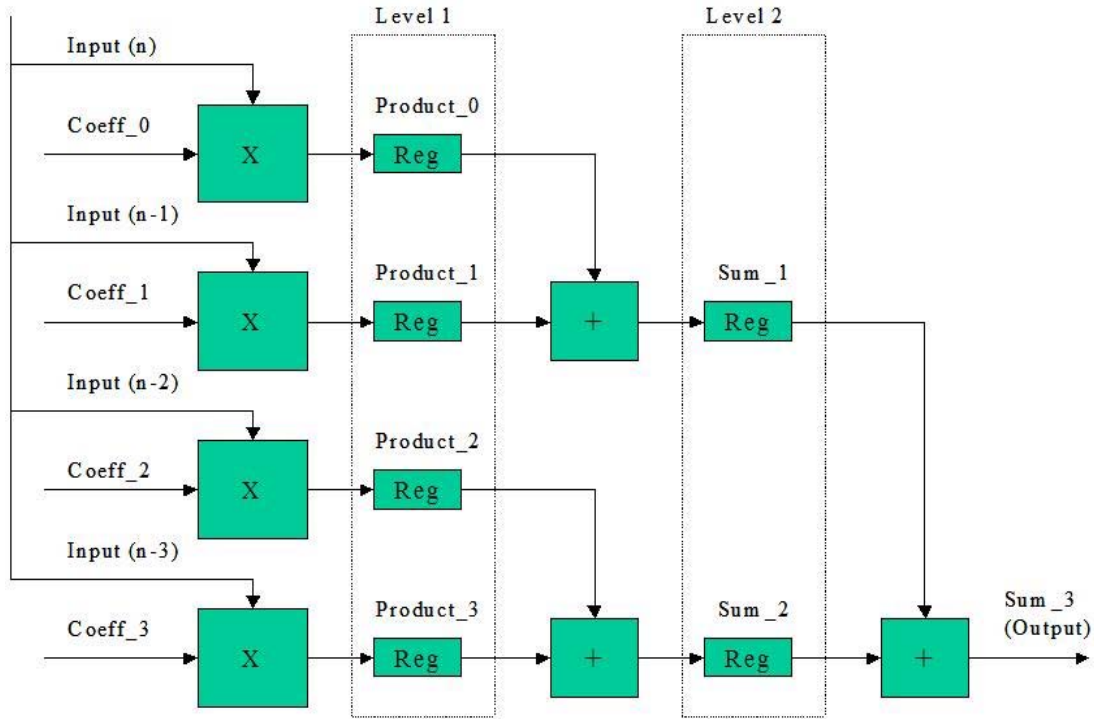
Figure 4.8: Block diagram of a pipelined FIR filter using multipliers, adders, and registers. Number of levels is logarithmic to the number of coefficients.

three adders and one clock-cycle of time, we would use one adder and three clock-cycles of time before moving on to the next set of values. Therefore, we would serve to greatly reduce the size of the physical circuit, but at the same time would greatly increase the time needed to accomplish the filtering. A block diagram of the sequential FIR filter used in our design is given below in Figure 4.9.

The Data Clocker unit has the filter coefficient values stored in it and will cycle through each of them before accepting a new Input value. Therefore, the entire circuit must cycle through its operations a number of times equal to the number of coefficients used in the filter, and hence the circuit will take that many clock cycles to complete the filtering for each input. The Sum register holds the value of the previously added numbers so that the total can again be added to on the next clock cycle. The Mux_sel

107

register holds a signal that tells the mux to use all zeros on the first addition of each cycle as it will be using a fresh input.
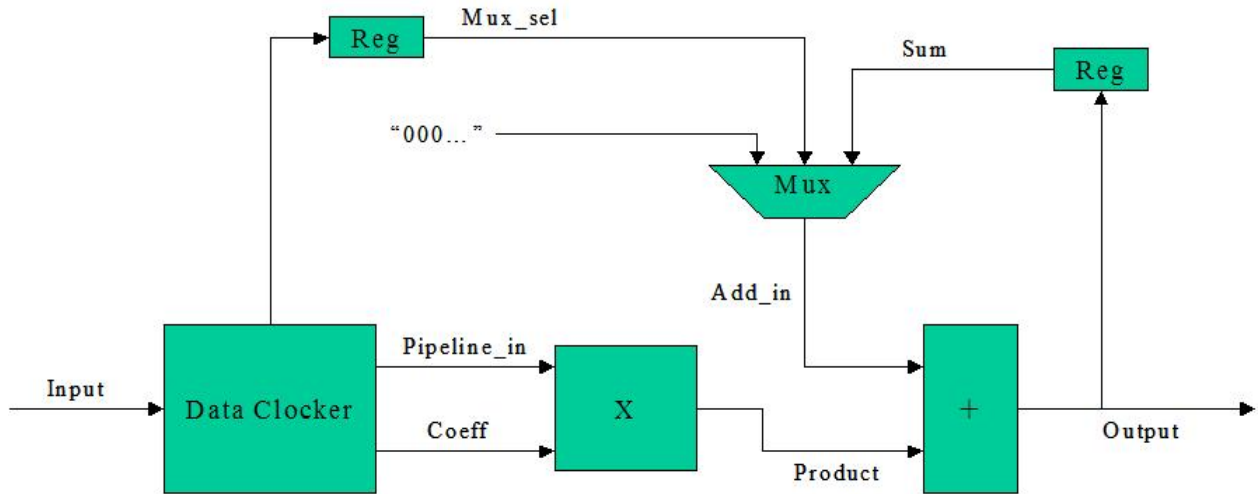
Figure 4.9: Block diagram of a sequential FIR filter using a clock unit, multiplier, adder, mux and registers. Data Clocker cycles through all stored coefficients before accepting a new Input value.

Creating a VHDL model of a structural, sequential design can become very complicated due to timing issues. A near-fully sequential template for the SSS is presented in Appendix O. The filter unit shown above must itself, use a number of clock cycles equal to the number of coefficients it has, plus one (for the registers). The Data Clocker keeps track of this cycle and sends out a 'ready' signal each time that is used as a clock by all other clocked devices in the filter's module. In the VHDL sequential template, a sequential multiplier is also implemented that requires a number of clock cycles equal to that of the maximum bit-width of its two inputs (plus one again for the registers). Therefore the generic variable 'extension' used in Appendix O corresponds to

the total number of clock cycles needed to execute its operations on each input value.
This value can be found with the equation:

$$extension = (\max[in_{bit\_width} \quad coeff_{bit\_width}] + 1)*(filt_{length} + 1) \qquad (4.3)$$

As in the pipelined VHDL code, the sequential template also uses the 'generate' function in some of it's components, as well as generics for variables and generic ports to tie everything to the top-level element (tsui_sss_converter_struct2_sequ_dig).

Now that we have VHDL templates for both a pipelined and a sequential SSS design, we need to be able to easily adjust them for new input, output, and coefficient bit-widths, and for new filter designs. Corresponding test bench templates were also created in order to automatically design test files for the SSS circuits. To accomplish this task, both Matlab and Perl programs will be used.

The first step is to use the 'coeff_gen.m' file that is presented in Appendix P (developed with help from [6]), in Matlab to create an output text file containing the desired filter information. Upon running the coeff_gen.m file, the user will be prompted to enter the desired filter length to be used in the SSS, as well as the total desired coefficient bit-width and the fractional bit-width (number of bits to the right of the decimal point). The user will also be prompted to input a testing input and sampling frequency, and an input bit-width value in order to create a list of formatted input values for use with a test bench. Matlab will then create an output file 'filter_coeff.txt' that contains the filter length, coefficient bit-width, and formatted lists of the coefficients to

109

be input directly into the VHDL templates. It will also create an output file 'tb_input.txt' that contains a formatted input list.

Once we have the 'filter_coeff.txt' file, we can combine it with our VHDL SSS template files to use as input to the Perl script 'gen_sss.prl' presented in Appendix Q (developed with help from [7]). In the command line for the Perl script, the user must define an output file name for the output '.stats' file, as well as the desired input bit-width, output bit-width, and desired VHDL design outputs (pipelined, sequential, abstract). Using the input information, the Perl script will replace the formatted word strings in the VHDL template files, with the correct numerical or bit-string values, or coefficient lists. New, formatted VHDL files will be created that can be used directly by simulation and synthesis tools to create layouts and tape-outs of the SSS design. The output '.stats' file will contain filter length and bit-width information of the design the user created.

To utilize the 'tb_input.txt' file, a second Perl script 'gen_tb.prl' was developed in parallel to 'gen_sss.prl'. This script uses the same basic process to create a usable VHDL test bench file for each VHDL SSS design that is created. 'tb_input.txt' is presented in Appendix R. This script only required the user to enter the input and output bit-widths in order to generate the test bench circuits.

4.4 Synthesis tools circuit generation results

In this final section, the results of running the SSSs designed in this chapter through various professional circuit synthesis tools are presented. This will verify the use

110

of the developed code in industry synthesis tools, for use with both FPGA and ASIC.

Area, speed, and power measurements will be analyzed. Four different input

combinations will be considered as are illustrated in Table 4.1.

The first synthesis we will use to verify the VHDL designs is Mentor Graphics'

Precision Synthesis for FPGAs. As the software name states, this program is to

synthesize FPGA implementations. The software will take a specified FPGA model, and

place and route the implemented code according to the limitations of that specific FPGA.

Using the Virtex II Pro option, the results are listed in Table 4.2. Graphical

representations of the size and speed results are presented in Figure 4.10 and Figure 4.11,

respectively.

Table 4.1: Input parameter combinations to be tested.

Parameter Set Designation #	Filter Length	Input Bit-Width	Coefficient Bit-Width	Output Bit-Width
1 [Small]	32 tap	4	4	12
2 [Medium]	32 tap	6	6	16
3 [Large]	32 tap	10	12	26
4 [Medium]	64 tap	6	6	16

Table 4.2: Synthesis results using Mentor Graphic' Precision RTL tool with Virtex II Pro (2VP2fg256) architecture. Frequency of 100 MHz.

Design Implementation	Design Combination	Latency	Speed	Size
		Data Arrival	*Samples / Second*	*Accumulated*

		Time (ns)		Instances
Pipelined	*1*	12.88	91,925,846	1,523
	2	17.42	79,590,676	9,055
	3	28.76	69,132,388	40,803
	4	18.55	80,111,087	15,131
Sequential	*1*	8.89	132,336	1,613
	2	8.89	94,525	2,234
	3	8.68	52,130	3,710
	4	8.68	49,873	3,537
Abstract	*1*	5.68	17,605,633	787
	2	4.83	14,788,524	5,207
	3	6.37	6,037,918	21,123
	4	5.84	14,677,103	8,563

The results are given in terms of the software parameters. The speed, or Data

Arrival Time (DAT), corresponds to the clock frequency of about 100MHz and is

therefore given in nano seconds. Area on the other hand is not a direct physical

measurement, but is rather a count of the number of instances required to implement the

code. These instances include physical digital devices such as D-flip/flops, latches,

MUXs, and external gates.

By analyzing the results from Table 4.2 we can see a number of trends that help

us to verify that our code in indeed working. First, notice that for each implementation

(pipelined, sequential, and abstract), the size of the design increases as the parameters of

the design are increased (use the design combination numbers 1-4 as reference). As

expected, the pipelined design increases in size much faster than the sequential because it

implements new components for each additional stage of processing, where as the
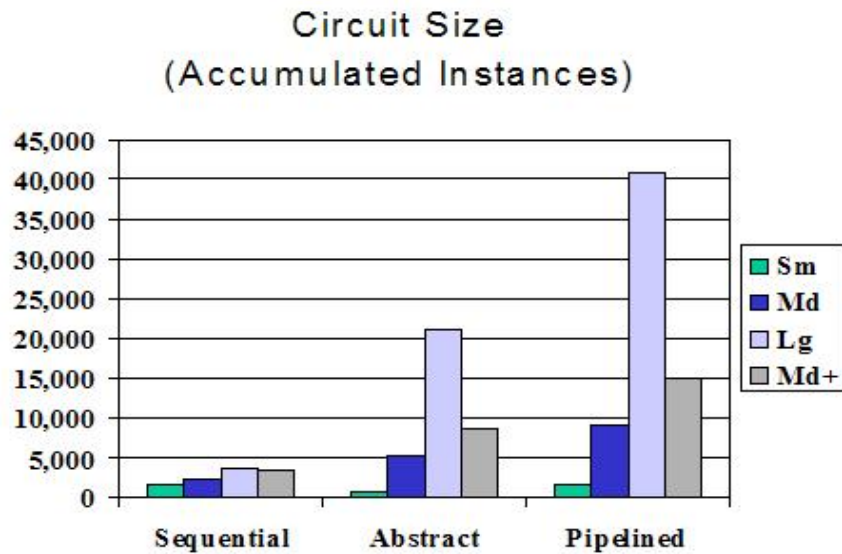
Figure 4.10: Circuit size results using Mentor Graphic' Precision RTL tool with Virtex II Pro (2VP2fg256) architecture. Frequency of 100 MHz.
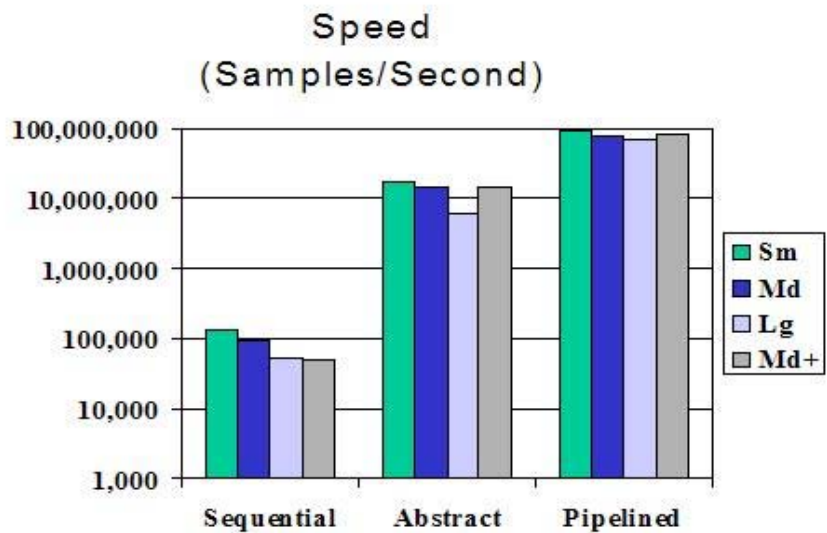
Figure 4.11: Circuit speed results using Mentor Graphic' Precision RTL tool with Virtex II Pro (2VP2fg256) architecture. Frequency of 100 MHz.

sequential will reuse the components. The abstract implementation seems to show a good combination of pipelining and sequencing as it's values tend to fall in between those of the two prior implementations.

Analyzing the latency results also gives expected results. The pipelined DAT is increased as more levels are added to the parameters, where as the DAT of the sequential implementations remain very similar. This is because the circuit is not increased in length for the sequential, but rather, just runs through the same sequence numerous times. To a casual observer, the pipelined design may seem slower, as it has a larger latency than the sequential, but while the pipelined design takes longer for a result to appear at the back end, once it does, it will have a new value each clock cycle. This can be seen in the speed results where the pipelined design is greater than two magnitudes faster than the sequential design. The sequential will produce a result quicker, but the 'correct' result will not appear until a number of cycles of the entire circuit have been accomplished. Again, the abstract implementation seems to show a partial use of sequential processes as it had the smallest DATs. The abstract version of the circuit, as expected, displays speed values between that of the pipelined and sequential.

The second synthesis we will use to verify the VHDL designs is using the Cadence Design Flow software bundle. The software package was used under the Wright State University license and included such programs functions as: mapping, constraining, floorplanning, power mapping, placement, clocktree design, optimized placement, and multiple routings. This synthesis is designed for ASIC circuits with 0.25 um technology. Results are presented in Table 4.3, and are given in the same format as

114

those in Table 4.2. Graphical representations of the size and speed results of the Cadence

synthesis are presented in Figure 4.12 and Figure 4.13, respectively.

Table 4.3: Synthesis results using Cadence design synthesis flow for 0.25 um technology.

Design Implementation	Design Combination	Latency	Speed	Size
		Data Arrival Time (ns)	*Samples / Second*	*Cell Area (um^2)*
Pipelined	*1*	9.69	124,882,922	155,344
	2	10.92	128,228,143	405,840
	3	13.38	134,918,585	1,994,901
	4	11.32	132,555,673	818,851
Sequential	*1*	2.11	556,513	21,948
	2	2.35	357,742	30,056
	3	3.44	131,690	50,471
	4	3.10	139,825	45,048
Abstract	*1*	3.45	29,078,220	43,703
	2	3.50	20,395,676	91,091
	3	5.11	7,534,091	350,667
	4	3.67	23,369,946	171,434

The same trends are present for the Cadence synthesis for ASIC, as were for the

MentorGraphics for FPGA. The size in this instance is measured in micro meters

squared rather than in simple logic instances, but same size trends apply here. As the

parameter sizes increase, so do the circuit sized. Again the pipelined design increased

much faster in size than does the sequential, with the abstract version falling at valued in-

between the pipelined and sequential designs.
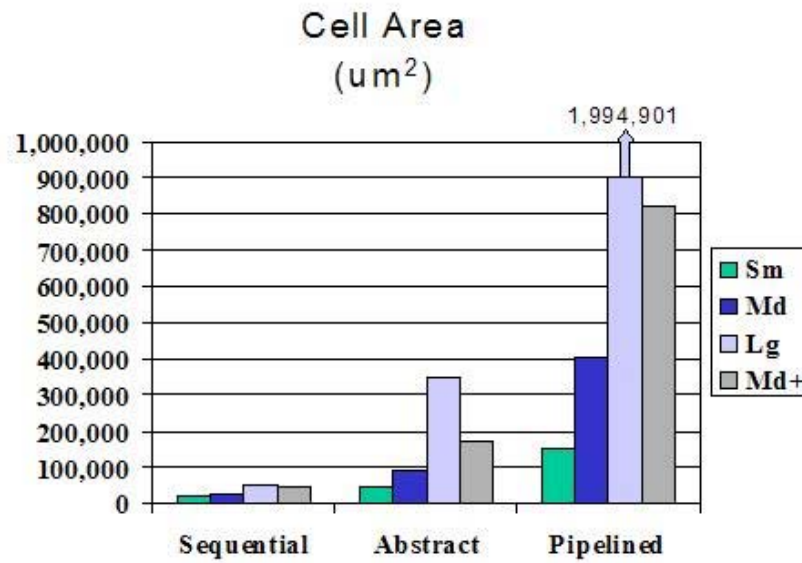
Figure 4.12: Circuit size results using Cadence design synthesis flow for 0.25 um technology.
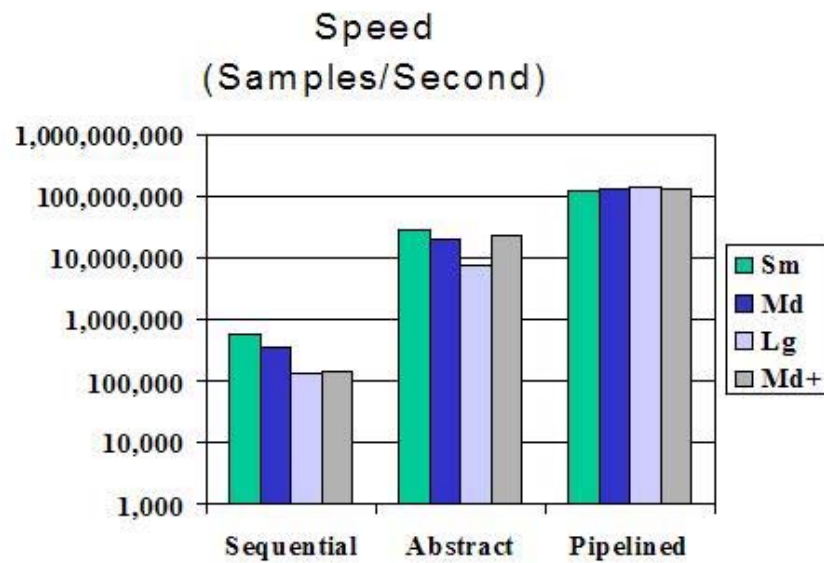
Figure 4.13: Circuit speed results using Cadence design synthesis flow for 0.25 um technology.

116

Latency measures show the pipelined circuit with the greatest initial DAT, followed by that of the abstract circuit, and the sequential circuit respectively. As in the case of the MentorGraphics synthesis, this can be deceiving if the latency is considered as the circuit's speed. This does not mean that the pipelined is the slowest circuit, it simply means that the pipelined circuit takes longer for the initial data to get from the input to the output.

The Cadence speed results again verify the expected results. The pipelined circuit is over two magnitudes faster than that of the sequential circuit, with the abstract circuit speed falling in-between that of the pipelined and sequential.

The final synthesis that was run was a power test using the web version of Altera's Quartus II simulator designed for FPGAs. For the simulation in this test, the Stratix II FPGA device option was utilized and the inputs to the circuit were randomly selected to simulate the unpredictable nature of sampled data in a real device. Both dynamic thermal power and static thermal power levels were measured. The numbers are given in Table 4.4. Graphical representations of the power results are presented in Figure 4.14 for dynamic values and Figure 4.15 for static values.

Before analyzing the power results, it is important to recognize that the largest pipelined parameter combination was not synthesizable using this FPGA type because it was too big to fit on the device. This is further support for the utilization of several design options. Such a situation could also occur when designing an ASIC if the required chip area is too small for the circuit to be synthesized onto.

Table 4.4: Power results using Altera Quartus II simulator for FPGAs--Stratix II
EP2S15F672C5.

Design Implementation	Design Combination	Thermal Power Dissipation	Thermal Power Dissipation
		Dynamic (mW)	*Static (mW)*
Pipelined	*1*	74.30	325.17
	2	124.28	328.14
	3	Na	Na
	4	148.05	329.08
Sequential	*1*	12.65	324.69
	2	11.63	325.27
	3	9.10	326.80
	4	11.60	325.56
Abstract	*1*	53.08	324.68
	2	126.52	327.30
	3	254.70	340.43
	4	135.41	328.23

Thermal power dissipation represents the power used in order to use the elements in a design. Therefore, the power dissipation value should correspond in a parallel manner to that of the size of the circuit being implemented. Looking at Figure 4.14, the power trend matches that of the size trend in the previous synthesis results. As the parameter sizes increase, so do the power results within each circuit design. The pipelined design has the greatest power utilization, while the sequential has the smallest. The abstract, as is expected, has a utilization less than the pipelined, but greater than the sequential.

The static thermal power dissipation is caused by the leakage of current through the gates on the FPGA and through the input and output pins. Since a specific FPGA will contain the same number of gates regardless of the circuit designed onto it, this
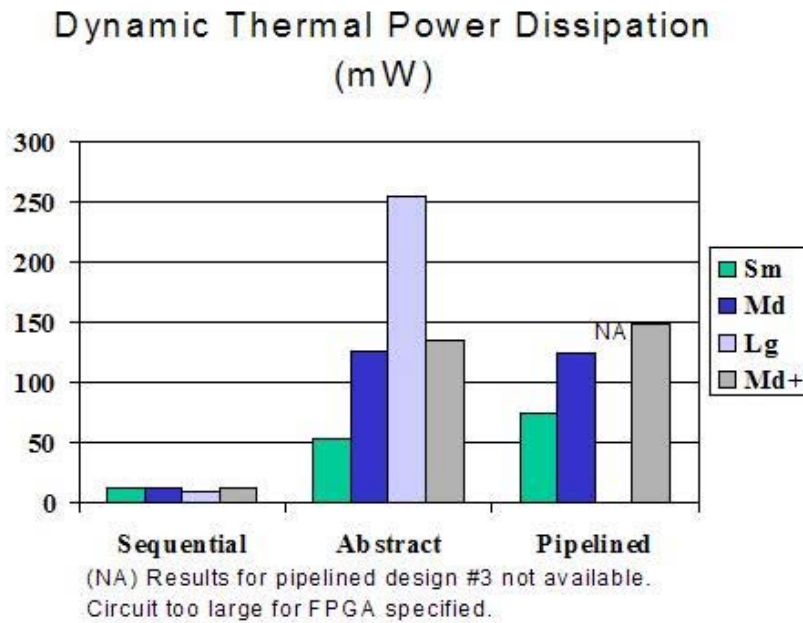
Figure 4.14: Dynamic thermal power dissipation using Altera Quartus II simulator for FPGAs--Stratix II EP2S15F672C5.
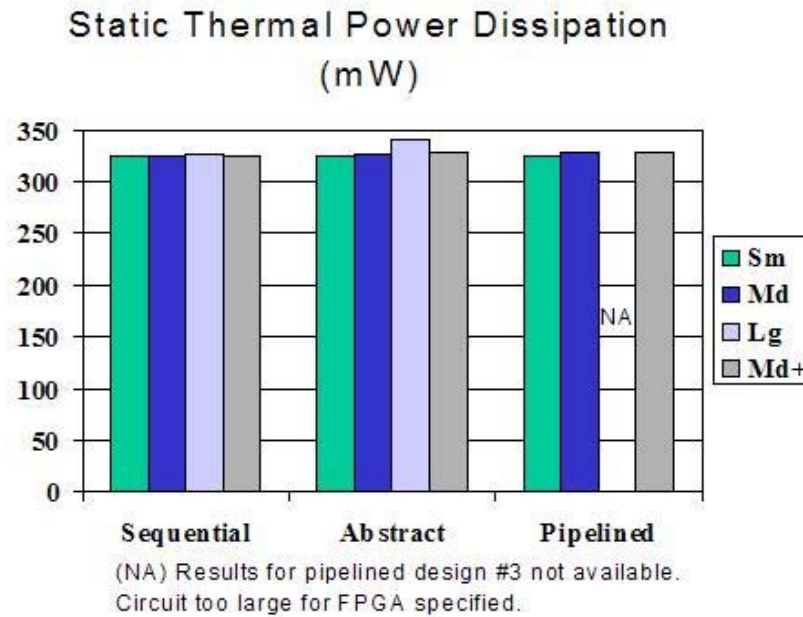
Figure 4.15: Static thermal power dissipation using Altera Quartus II simulator for FPGAs--Stratix II EP2S15F672C5.

value should not be altered greatly by different design implementation. As can be seen in Figure 4.15, all static reading are very close in magnitude. They vary slightly due to increasing parameter values which will cause greater numbers of input and output pins in larger designs.

4.5 Chapter Summary

In this chapter results were presented of all three research objectives. The chapter started by analyzing the significant trends of I/Q imbalances when varying input parameters. Such trends as safe input bit-width values and frequency proportionality can be useful in designing SSS circuits with minimal imbalances.

Next the coding process was reviewed to develop and implement the SSS into useful, abstract Matlab, VHDL, and Perl files. Using the code developed, a designer can easily change parameters such as filter lengths and bit-widths to accommodate imbalance trends.

Finally, an analysis of the results of running VHDL files using specific design parameters, and specific optimization levels (pipelined, sequential, and abstract) through professional synthesis softwares was done. The trends found from these syntheses were identified and discussed.

4.6 References

[1]   Filter Design HDL Coder. "What is the Filter Design HDL coder?" Acquired from MATLAB help options, 1994-2005 The MathWorks, Inc. 10 October 2005.

[2]   Filter Design HDL Coder. "FDATool Plug-in—the GUI." Acquired from MATLAB help options, 1994-2005 The MathWorks, Inc. 10 October 2005.

[3]   Armstrong, James A. and Gray, Gail. *VHDL Design—Representation and Synthesis.* 2nd Ed. Prentice-Hall, Inc, 2000.

[4]   Perry, Douglas L. *VHDL—Programming by Example.* 4th Ed. The McGraw-Hill Companies, Inc. 2002.

[5]   Patterson, David A. and Hennessy, John L. *Computer Organization And Design.* 3rd Ed. Morgan Kaufmann Publishers, August 2004.

[6]   MATLAB Function Reference. "fprintf." Acquired from MATLAB help options, 1994-2005 The MathWorks, Inc. 02 February 2006.

[7]   http://www.htmlite.com/. Various references used. 09 February 2006.

V. Analysis and Conclusions

5.1 Explanation of the Problem

The problem to be solved was to accomplish the characterization and implementation of a SSS for I/Q downconversion utilizing top-level, portable design strategies. AFRL has a great use for receiver systems that can utilize both I/Q downconversions, and digital signals. Such a system could also benefit any number of other government or industry designers. This has not been possible in the past due to the speed limitations of using digital hardware. Recently, digital technology has increasing potential in speed and size, making digital I/Q sampling downconversion a legitimate option.

One specific method of digital I/Q downconversion is that of the SSS presented in this research. In order to make such a design useful however, the imbalance of the I and Q data streams needs to be acceptable. Since this sampling method is not well-developed or documented yet, classification of the effects of input parameters on such imbalances is vital knowledge to have.

With acceptable parameter knowledge, effective designs for SSS circuits can be developed using abstract HDL and Matlab codes. Scripts can be used to make the design process easier on the user. These designs can then be used with professional design synthesis software programs to create physical circuit representations. Each of these objectives was successfully completed in this research and are summarized in the following sections.

5.2 Summary of Literature Review

A review of a variety of book, paper, and internet sources was done in order to investigate and verify the methods and current technologies that support this research. The basis behind the SSS for digital downconversion presented by James Tsui was reviewed in detail along with digital filtering methods. An alternative method of I/Q digital downcoversion (using the Hilbert transform) was also explored and the differences between these alternatives and the SSS discussed. Using Matlab tutorials and through personal discussions with industry professionals, methods of modeling data imbalances were verified and presented as well. Finally, a comprehensive study of the ASIC/FPGA design and synthesis process was conducted to give insight on the significance of an abstract design model, and show the need for optimization within a hardware design. This studies' design process of the SSS digital downconverter stems from basic principles presented in this literature review.

5.3 SSS Characterization: Significance, Limitations, and Further Research

The first step towards meeting the design objectives of this research was to significantly characterize parameter effects of hardware on the SSS. By analyzing the SSS scheme using Matlab simulations and data manipulation, it was found that the effect of certain parameters on amplitude and phase imbalances of the I/Q data streams varies.

- Filter length has a limited effect on imbalances. In all but the very worst
  cases, the effect of using any range of filter lengths only seems to affect the
  overall imbalances by less than 0.005%.

- Sampling size has an increasingly lower effect on imbalances as it increases.
  In a real system, the sampling size theoretically nears infinity, so it should be
  a null factor.

- Coefficient bit-width has a consistent effect on imbalances, therefore
  variances in it will should have a limited effect on overall imbalances.

- Input bit-widths have a very sporadic effect on imbalances when they are
  small. This effect varies greatly with different input/sampling frequency
  combinations. Imbalance values level off once the bit-width reaches a certain
  size. Bit-width variations above the leveled off value should have a limited
  effect on overall imbalances. See Appendix D for leveled off values.

- Output bit-widths present the same characteristics as the input bit-widths.

The largest effect on I/Q imbalances stems from variations in input and output bit-
widths. Small bit-width values can cause very significant imbalances to occur, therefore
limiting the usefulness of the SSS. Since these imbalances will vary depending on the
input and sampling frequency, a design that will be used to sample multiple frequencies,
or will use multiple sampling frequencies will need to either use bit-widths above all
leveled off values of Appendix D, or will need to include an imbalance correction module
at the back end. Either option will lend towards limiting the size and speed of the circuit.
A unit that will be used at a pre-specified input and sampling frequency, can potentially

be designed to use a smaller input/output combination that will have acceptable imbalances without the use of a correction module.

This initial classification research lends itself toward a couple further research options. The first option would be to further analyze the effect of 'combined' parameter change effects on I/Q imbalances, and to develop a system or algorithm to further optimize the designs for a range of input/sampling frequencies. A second option would be to explore methods of digital imbalance correction, and integrate it into SSS scheme design.

5.4 HDL Code Development: Significance, Limitations, and Further Research

Once the SSS was characterized, the second step of this study was to develop a portable, top-level set of hardware codes. Utilizing several software programs, development of a usable, abstract design methodology for implementing the SSS into VHDL codes was accomplished. Matlab proved to be a useful tool in developing filter designs and organizing filter coefficients for use in the VHDL code. VHDL syntax operations such as the 'generate' command made it possible to create multiple modules as needed for specified design variables, and the 'generic' variable type declarations within VHDL made it possible to pass top-level parameters such as bit-widths down to lower level modules of the design. Finally, Perl scripting let us put the Matlab process together with the VHDL template codes to make a user friendly interface to create custom SSS designs and cooresponding test benches with near minimal effort.

There are a number of directions that could be pursued to further this design

process. The most obvious way to advance the process would be to mature the Perl code.

Currently, a user is required to run both a matlab script, and then use the output with

VHDL templates and a Perl script to get a final SSS VHDL design. There is potential for

this process to be matured to run the Matlab automatically from the Perl script, and even

to divide the VHDL SSS templates into modular component templates so that they can be

more flexibly constructed in an automatic process in Perl. A second possible extension

of this design is to increase the breadth of the optimization effort by including more

template options. This 'slider-bar' approach would allow a user to pipeline different

levels (two-stage pipeline, three-stage pipeline, etc.) of the design in order to manipulate

the size versus speed trade offs to fit requirements in both specifications.

One final extension research project that could improve the bit-width optimization

of the SSS system would be a system or algorithm to optimize a filter length with respect

to coefficient bit-widths. For example, a filter of length 32 will have coefficient values

that are too small at each end to be represented by a 6-bit signed representation. This will

result in unnecessary 'zero' valued coefficients. Therefore in this case it would be better

to drop the coefficient to reduce the size of the circuit, or to re-calculate the coefficients

to a shorter filter length.

5.5 Design Synthesis: Significance, Limitations, and Further Research

A final step to this research was to validate the VHDL code for synthesis on

multiple platforms of professional software. Successful synthesis runs were made on

various tools including MentorGraphics' Precision RTL, MentorGraphics' Leonardo Spectrum, and the Cadence Synthesis Suit.

The design synthesis of the SSS design code developed in this research revealed some trends as to the extent of the effect of using pipelined designs, sequential designs, and top-level abstract (behavioral) designs on the size, speed, and power of the circuit. Common trends include that the size of a design increases as the parameter size increases, while the speed decreases with a parameter size increase. Also found, was that synthesis differed from software to software, and with different technology goals (ASIC vs. FPGA). Pipelined designs acted as expected by have larger physical architectures than sequential designs, but also displaying longer initial DATs. Abstract implementations either took the same form at the pipelined, or showed a combination of pipelined and sequential implementations.

Continued research on design optimized synthesis of the SSS method could consist of mixing and matching different professional software tools in an attempt to find better optimization. Extended analysis of which circuit parameters (filter size, input bit-width, output bit-width, coefficient bit-width) have the largest affect on size, speed, and power parameters could improve optimization before even running professional tools.

Appendix A: Truncation Function (MATLAB)

```
%***** Matlab function for Jesse P Somann thesis *****
%***** last edited 28 Jan 06 *****
%***** Air Force Institute of Technology *****

% This function takes a number(or array) of set precision, and translates
% it to a truncated binary equivalent -- user can set binary precision.

function [funct_out] = trunk_down(input, pos_dec_in, neg_dec_in)

fpt = fi(input, 1, pos_dec_in + neg_dec_in, neg_dec_in);

funct_out = double(fpt);
```

Appendix B: Special Sampling Scheme (MATLAB)

```matlab
%***** Matlab Code for Jesse P Somann thesis *****
%***** Original by James Tsui--last edited 29 Nov 05 *****
%***** Air Force Institute of Technology *****

%df8_16.m I/QBK generates I and Q channels from fs/2
clear
format long;

n1 = 1024;        % number of samples to be taken
n = [0:n1-1];
f = input('enter input frequency in MHz = ');
sf = input('enter input frequency in MHz, must be at least twice f = ');
% fb = input ('enter input of second frequency for dual setup as above or 0
% if none = ');
fb = 0*10^6;

%form sine wave
x = sin(2*pi*(f*10^6)*((1/sf)*1e-6)*n+.01)+sin(2*pi*fb*1e-8*n+.01);

% truncated values
x_2 = trunk_down(x,1,1);
x_3 = trunk_down(x,1,2);
x_4 = trunk_down(x,1,3);
x_6 = trunk_down(x,1,5);
x_8 = trunk_down(x,1,7);
x_12 = trunk_down(x,1,11);
figure(2);
plot(n,x,n,x_2,n,x_3,n,x_4,n,x_6,n,x_8,n,x_12);
axis([20 50 -1.1 1.1])

%************* generate I and Q channels***********
xx = reshape(x,4,n1/4);
xrm = [xx(1,1:n1/4); -xx(3,1:n1/4)];
xim = [xx(2,1:n1/4); -xx(4,1:n1/4)];
xr = reshape(xrm,1,n1/2);
xi = 1*reshape(xim,1,n1/2);

%********** create a low pass filter *************
dbst = 70;
betat = .1102*(dbst-8.7);  % if dbst is > 50
%betat = .5842*(dbst-21)^0.4+.07886*(dbst-21);  % if 50 >= dbst >= 21
%betat = 0;  % if dbst is < 21
windowt = kaiser(32,betat);
figure(2);
wvtool(windowt)
hkt = fir1(31,0.5, windowt);
figure(3);
freqz(hkt,1,512)
filtab = reshape(hkt,2,16);
filtat = filtab(1,1:16);
filtbt = filtab(2,1:16);

%********* filtering ***********************
cnvlra = conv(filtat,xr);
cnvlrb = conv(filtbt,xr);
cnvlia = conv(filtat,xi);
```

```matlab
cnvlib = conv(filtbt,xi);
I1 = [cnvlra; cnvlrb];
It = reshape(I1,1,2*length(cnvlra));
Q1 = [cnvlia; cnvlib];
Qt1 = reshape(Q1,1,2*length(cnvlra));
Qt = [0 Qt1(1:n1+30-1)];   %add 0 as the first point

%*** Channel Anaylsis *****(R = amplitude of fft, theta = angle of fft)
Y = fft(x,n1); R = abs(Y); theta = angle(Y);
YI = fft(It(15:n1-15),n1); RI = abs(YI); thetaI = angle(YI);
YQ = fft(Qt(15:n1-15),n1); RQ = abs(YQ); thetaQ = angle(YQ);

%***Channel Comparisons***
R_diff = RI - RQ;
R_imbalance = (R_diff)./RI.*100;
theta_diff = (thetaI-thetaQ)./pi.*180;
theta_imbalance = (90 - theta_diff)./90.*100;

%*** Power/Frequency ***** (Power signals of fft, should spike at main
frequency
Py = Y.* conj(Y) / n1;
PyI = YI.* conj(YI) / n1;
PyQ = YQ.* conj(YQ) / n1;

%*** Finds the spot of amplitude spur of fft ****
[max_amp,max_x] = max(RI(round(n1/100):n1/2));  %rounds off first values
max_x = max_x + round(n1/100) - 1;              %compensates for rounding

f = 100*(0:n1/2)/n1;

%************* plot I/Q Channels ************
figure(5);
subplot(3,2,1)
plot(1:n1,real(YI))
title('Real I Channel')
subplot(3,2,2)
plot(1:n1,imag(YI))
title('Imaginary I Channel')
subplot(3,2,3)
plot(1:n1,real(YQ))
title('Real Q Channel')
subplot(3,2,4)
plot(1:n1,imag(YQ))
title('Imaginary Q Channel')
subplot(3,2,5)
plot(1:n1,real(Y))
title('Real Original Channel')
subplot(3,2,6)
plot(1:n1,imag(Y))
title('Imaginary Original Channel')

%************* plot I/Q Power Bins ************
figure(6);
subplot(3,1,1)
plot(f,PyI(1:(n1/2+1)))
% axis([0 600 0 30])
title('Frequency content of I Channel')
xlabel('frequency (MHz)')
subplot(3,1,2)
plot(f,PyQ(1:(n1/2+1)))
% axis([0 600 0 30])
```

```matlab
title('Frequency content of Q Channel')
xlabel('frequency (MHz)')
subplot(3,1,3)
plot(f,Py(1:(n1/2+1)))
title('Frequency content of Original Signal')
xlabel('frequency (MHz)')

%************* plot I/Q Characteristics ************
figure(7);
subplot(3,2,1)
stem(1:n1,RI)
title('Amplitude I Channel')
subplot(3,2,2)
plot(1:n1,thetaI)
title('Angle I Channel')
subplot(3,2,3)
stem(1:n1,RQ)
title('Amplitude Q Channel')
subplot(3,2,4)
plot(1:n1,thetaQ)
title('Angle Q Channel')
subplot(3,2,5)
stem(1:n1,R)
title('Amplitude Original Channel')
subplot(3,2,6)
plot(1:n1,theta)
title('Angle Original Channel')

%************* plot I/Q Imbalances ************
figure(8);
subplot(2,2,1)
plot(1:n1,theta_diff);
title('I/Q Phase Difference (Degrees)')
subplot(2,2,2)
plot(1:n1,R_diff)
title('I/Q Amplitude Difference')
subplot(2,2,3)
plot(1:n1,theta_imbalance);
title('I/Q Phase Imbalance (%)')
subplot(2,2,4)
plot(1:n1,R_imbalance)
title('I/Q Amplitude Imbalance (%)')

%************** plot the results ***************
figure(1);
subplot(2,1,1)
plot(n,x);
subplot(2,1,2)
plot(1:n1+30,Qt, 1:n1+30,It)
axis([0 n1+50 -1 1])
xlabel('Time sample')
ylabel('Outputs of I Q channels')
```

131

Appendix C: Bit-Width Optimizer (MATLAB)

```matlab
%***** Matlab Code for Jesse P Somann thesis *****
%***** Original by James Tsui--last edited 29 Nov 05 *****
%***** Air Force Institute of Technology *****

%df8_16.m I/QBK generates I and Q channels from fs/2
clear
format long;

n1 = 2048;        % number of samples to be taken
n = [0:n1-1];
% f = f_in;        % input frequency from function call
% sf = sf_in;      % sampling frequency from function call
f = input('enter input frequency in MHz = ');
sf = input('enter input frequency in MHz, must be at least twice f = ');
% fb = input ('enter input of second frequency for dual setup as above or 0
% if none = ');
fb = 0*10^6;

x = sin(2*pi*(f*10^6)*((1/sf)*1e-6)*n+.01)+sin(2*pi*fb*1e-8*n+.01);    %form
sine wave
% figure(1);
% plot(n,x);


list = 5;

for set = 1:1:list
    amp_diff_opt(set) = 100;
    amp_diff_opt_phase(set) = 100;
    phase_diff_opt(set) = 100;
    phase_diff_opt_amp(set) = 100;
    max_bit_in_opt(set) = 100;
    max_bit_coeff_opt(set) = 100;
    max_bit_out_opt(set) = 100;
    max_bit_in_opt_p(set) = 100;
    max_bit_coeff_opt_p(set) = 100;
    max_bit_out_opt_p(set) = 100;
end

max_bit_in = input('enter maximum input bit width = ');
max_bit_coeff = input('enter maximum coefficient bit width = ');
max_bit_out = input('enter maximum output bit width = ');

max_amp_imb = input('enter maximum amplitude imbalance allowable in % = ');
max_phase_imb = input('enter maximum phase imbalance allowable in % = ');

for cnt_in = 2:1:max_bit_in                    %loops throught input bit widths
1:max_bit
    for cnt_coeff = 2:1:max_bit_coeff          %loops throught coefficient bit
widths 1:max_bit
        for cnt_out = 2:1:max_bit_out          %loops throught output bit widths
1:max_bit

            x_trunk = trunk_down(x,1,cnt_in-1);    %trunkate the input bit
width

            % figure(2);
```

```matlab
            % plot(n,x,n,x_2,n,x_3,n,x_4,n,x_6,n,x_8,n,x_12);
            % axis([20 40 -1.1 1.1])

            %************* generate I and Q channels***********
            xx = reshape(x_trunk,4,n1/4);
            xrm = [xx(1,1:n1/4); -xx(3,1:n1/4)];
            xim = [xx(2,1:n1/4); -xx(4,1:n1/4)];
            xr = reshape(xrm,1,n1/2);
            xi = 1*reshape(xim,1,n1/2);

            %*********** create a low pass filter *************
            dbst = 70;
            betat = .1102*(dbst-8.7);  % if dbst is > 50
            %betat = .5842*(dbst-21)^0.4+.07886*(dbst-21);  % if 50 >= dbst >=
21
            %betat = 0;  % if dbst is < 21
            windowt = kaiser(32,betat);          %use constant value for kaiser
window (32)
            % figure(2);
            % wvtool(windowt)
            hkt = fir1(31,0.5, windowt);         %fir filter command for kaiser
window
            hkt_trunk = trunk_down(hkt,1,cnt_coeff-1);    %trunkate the coeff
bit width
            % figure(3);
            % freqz(hkt,1,512)
            filtab = reshape(hkt_trunk,2,16);          %remix & shape filter
coefficients
            filtat = filtab(1,1:16);
            filtbt = filtab(2,1:16);

            %********** filtering ************************
            cnvlra = conv(filtat,xr);
            cnvlrb = conv(filtbt,xr);
            cnvlia = conv(filtat,xi);
            cnvlib = conv(filtbt,xi);
            I1 = [cnvlra; cnvlrb];
            It = reshape(I1,1,2*length(cnvlra));
            Q1 = [cnvlia; cnvlib];
            Qt1 = reshape(Q1,1,2*length(cnvlra));
            Qt = [0 Qt1(1:n1+30-1)];  %add 0 as the first point
            It_trunk = trunk_down(It,1,cnt_out-1);    %trunkate the I bit width
            Qt_trunk = trunk_down(Qt,1,cnt_out-1);    %trunkate the Q bit width

            %*** Channel Anaylsis *****(R = amplitude of fft, theta = angle of
fft)
            Y = fft(x,n1); R = abs(Y); theta = angle(Y);
            YI = fft(It_trunk(15:527),n1); RI = abs(YI); thetaI = angle(YI);
            YQ = fft(Qt_trunk(15:527),n1); RQ = abs(YQ); thetaQ = angle(YQ);

            %*** Channel Comparisons ***
            R_diff = RI - RQ;
            R_imbalance = (R_diff)./RI.*100;
            theta_diff = (thetaI-thetaQ)./pi.*180;
            theta_imbalance = (90 - abs(theta_diff))./90.*100;

            %*** Power/Frequency ***** (Power signals of fft, should spike at
main
            % frequency
            Py = Y.* conj(Y) / n1;
            PyI = YI.* conj(YI) / n1;
```

```matlab
            PyQ = YQ.* conj(YQ) / n1;

            %*** Finds the spot of amplitude spur of fft ****
            [max_amp,max_x] = max(RI(round(n1/100):n1/2));  %rounds off first
values
            max_x = max_x + round(n1/100) - 1;              %compensates for
rounding

            %*** Create array of imbalance measurements ***
            RI_var_this = abs(RI(max_x));
            R_diff_var_this = abs(R_diff(max_x));
            if R_diff_var_this == 0
                R_imbalance_var_this = 0;
            else
                R_imbalance_var_this = abs(R_imbalance(max_x));
            end
            if abs(theta_diff(max_x)) > 250
%newly added*****
                    theta_diff_var_this = abs(theta_diff(max_x)) - 180;
                    theta_imbalance_var_this = abs((90 -
abs(theta_diff_var_this))./90.*100);
            else
                theta_diff_var_this = theta_diff(max_x);
                theta_imbalance_var_this = abs(theta_imbalance(max_x));
            end

            done_amp = 0; done_ang = 0;
            for set2 = 1:1:list
                if R_imbalance_var_this > 0 && R_imbalance_var_this <
max_amp_imb && theta_imbalance_var_this < max_phase_imb && R_imbalance_var_this
< amp_diff_opt(set2) && done_amp == 0
                        for set3 = list:-1:set2+1
                            amp_diff_opt(set3) = amp_diff_opt(set3-1);
                            amp_diff_opt_phase(set3) = amp_diff_opt_phase(set3-1);
                            max_bit_in_opt(set3) = max_bit_in_opt(set3-1);
                            max_bit_coeff_opt(set3) = max_bit_coeff_opt(set3-1);
                            max_bit_out_opt(set3) = max_bit_out_opt(set3-1);
                        end
                        amp_diff_opt(set2) = R_imbalance_var_this;
                        amp_diff_opt_phase(set2) = theta_imbalance_var_this;
                        max_bit_in_opt(set2) = cnt_in;
                        max_bit_coeff_opt(set2) = cnt_coeff;
                        max_bit_out_opt(set2) = cnt_out;
                        done_amp = 1;
                end

                if theta_imbalance_var_this > 0 && R_imbalance_var_this <
max_amp_imb && theta_imbalance_var_this < max_phase_imb &&
theta_imbalance_var_this < phase_diff_opt(set2) && done_ang == 0
                        for set3 = list:-1:set2+1
                            phase_diff_opt(set3) = phase_diff_opt(set3-1);
                            phase_diff_opt_amp(set3) = phase_diff_opt_amp(set3-1);
                            max_bit_in_opt_p(set3) = max_bit_in_opt_p(set3-1);
                            max_bit_coeff_opt_p(set3) = max_bit_coeff_opt_p(set3-
1);
                            max_bit_out_opt_p(set3) = max_bit_out_opt_p(set3-1);
                        end
                        phase_diff_opt(set2) = theta_imbalance_var_this;
                        phase_diff_opt_amp(set2) = R_imbalance_var_this;
                        max_bit_in_opt_p(set2) = cnt_in;
                        max_bit_coeff_opt_p(set2) = cnt_coeff;
```

134

```matlab
                        max_bit_out_opt_p(set2) = cnt_out;
                        done_ang = 1;
                    end
                end

            end
        end
end

name1 = ['    Optimized amp  --  ' 'input width  --  ' 'coefficient width  --  '
'output width  --  ' 'Cooresponding phase']
list1 = [amp_diff_opt' max_bit_in_opt' max_bit_coeff_opt' max_bit_out_opt'
amp_diff_opt_phase']

name2 = ['    Optimized phase  --  ' 'input width  --  ' 'coefficient width  --
' 'output width  --  ' 'Cooresponding amp']
list2 = [phase_diff_opt' max_bit_in_opt_p' max_bit_coeff_opt_p'
max_bit_out_opt_p' phase_diff_opt_amp']
```

Appendix D: Safe Bit-width Values to Use

Input Frequency (fi) in MHz	Sampling Frequency (fs) in MHz	Input Bit-Width for Phase Imbalance	Input Bit-Width for Amplitude Imbalance	Output Bit-Width for Phase Imbalance	Output Bit-Width for Amplitude Imbalance	Coeff. Bit-Width for Phase Imbalance	Coeff. Bit-Width for Amplitude Imbalance
10	1000	6	5	7	6	3	3
20	1000	6	9	7	7	3	3
30	1000	5	6	7	7	3	3
40	1000	3	4	3	3	3	3
50	1000	8	6	9	8	3	3
60	1000	6	7	8	7	3	3
70	1000	5	6	8	9	3	3
80	1000	3	3	4	3	3	3
90	1000	5	6	7	9	3	3
100	1000	9	8	9	7	3	3
110	1000	5	5	7	9	3	3
120	1000	4	3	2	3	3	3
130	1000	5	6	8	9	3	3
140	1000	4	7	4	8	3	3
150	1000	8	10	7	10	3	3
160	1000	2	5	5	6	3	3
170	1000	5	6	7	9	3	3
180	1000	4	9	6	8	3	3
190	1000	5	5	6	9	3	3
200	1000	2	3	3	5	3	3
210	1000	5	6	7	9	3	3
220	1000	4	9	4	7	3	3
230	1000	2	5	6	9	3	3
240	1000	2	5	2	6	3	3
250	1000	7	7	8	8	3	3
260	1000	6	7	4	7	3	3
270	1000	5	5	7	7	3	3
280	1000	2	4	3	3	3	3
290	1000	5	6	6	7	3	3
300	1000	9	9	9	9	3	3
310	1000	5	5	7	9	3	3
320	1000	3	3	3	4	3	3
330	1000	5	6	6	7	3	3
340	1000	6	9	6	8	3	3
350	1000	6	10	8	10	3	3

360	1000	3	5	3	3	3	3
370	1000	5	6	6	7	3	3
380	1000	6	9	6	7	3	3
390	1000	5	5	7	9	3	3
400	1000	2	2	3	6	3	3
410	1000	5	6	6	9	3	3
420	1000	6	9	6	8	3	3
430	1000	5	6	7	9	3	3
440	1000	4	2	4	4	3	3
450	1000	6	9	10	8	3	3
460	1000	6	9	7	8	3	3
470	1000	5	6	8	7	3	3
480	1000	2	4	3	6	3	3
490	1000	5	5	7	6	3	3
498	1000	5	5	7	9	3	3
> 498 502<	1000	Large imbalance	Large imbalance	Large imbalance	Large imbalance	Large imbalance	Large imbalance
1250	3000	7	6	10	10	3	3

Appendix E: IQgen (VHDL)

```vhdl
-- ----------------------------------------------------------
-- Module: I/Qgen
-- Generated by Jesse P Somann
-- Started on 11 Oct 05
-- Finished on __
-- Purpose: to split the input data into a real (I) and
--  imaginary (Q) stream initiative to be convolved
-- ----------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

ENTITY I/Qgen IS
   PORT( clk                            :   IN    std_logic;
         clk_enable                     :   IN    std_logic;
         reset                          :   IN    std_logic;
         data_in                        :   IN   real; -- double
         dataI_out                      :   OUT   real;  -- double
         dataQ_out              :   OUT   real
         );

END I/Qgen;


------------------------------------------------------------------
--Module Architecture: I/Qgen
------------------------------------------------------------------
ARCHITECTURE rtl OF I/Qgen IS

-- Local Functions
--  Signals
  SIGNAL output_registerI                : real := 0.0; -- double
  SIGNAL output_registerQ                : real := 0.0; -- double

  SIGNAL counter4                   : std_logic_vector(1 downto 0);
  SIGNAL inverted                   : real := 0.0;

BEGIN

  Output_Register_process : PROCESS (clk, reset)
  BEGIN

    IF reset = '1' THEN
      output_registerI <= 0.0000000000000000E+000;
      output_registerQ <= 0.0000000000000000E+000;
      counter4 <= "00";

    ELSIF clk'event AND clk = '1' THEN

      IF clk_enable = '1' THEN

        CASE counter4 is
             when "00" =>
               output_registerI <= data_in;
             when "01" =>
```

138

```vhdl
                output_registerQ <= data_in;
            when "10" =>
                output_registerI <= data_in * (-1.0);
            when "11" =>
                output_registerQ <= data_in * (-1.0);
            when others =>
                output_registerI <= 0.0;
                output_registerQ <= 0.0;
        END CASE;
        counter4 <= counter4 + "01";

      END IF;
    END IF;
  END PROCESS Output_Register_process;

  -- Assignment Statements
  dataI_out <= output_registerI;
  dataQ_out <= output_registerQ;

END rtl;
```

```
----------------------------------------------------------------
-- Module: I/Qfir_filtat
-- Original genertion by MATLAB(R) 7.0.4 and the Filter Design HDL Coder 1.2.
-- Edit and Update by Lt Jesse Somann: 2006-01-20
-- ----------------------------------------------------------
-- HDL Code Generation Options:
--
-- TargetLanguage: VHDL
-- Name: I/Qfir_filtat
-- TargetDirectory: C:\Documents and Settings\Owner\Desktop\AFIT
classes\Thesis\Matlab\hdlsrc
-- LoopUnrolling: On
-- InlineConfigurations: Off
-- SafeZeroConcat: Off
-- TestBenchName: I/Qfir_filtat_tb
-- TestBenchStimulus: impulse step ramp chirp noise
--
-- Filter Settings:
--
-- Discrete-Time FIR Filter (real)
-- ------------------------------
-- Filter Structure  : Direct-Form FIR
-- Filter Length     : 16
-- ----------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;

ENTITY I/Qfir_filtat IS

   GENERIC(  max_flength         :   natural := 16);

   PORT( clk                            :   IN    std_logic;
         clk_enable                     :   IN    std_logic;
         reset                          :   IN    std_logic;
         filter_in                      :   IN    real; -- double
         filter_out                     :   OUT   real  -- double
         );

END I/Qfir_filtat;


----------------------------------------------------------------
--Module Architecture: I/Qfir_filtat
----------------------------------------------------------------
ARCHITECTURE rtl OF I/Qfir_filtat IS
  -- Local Functions

  -- Type Definitions
  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>) OF real; -- double
  TYPE coeff_pipeline_type IS ARRAY (NATURAL range <>) OF real; -- double

  -- Constants
  CONSTANT coeff                   : coeff_pipeline_type(1 TO max_flength) := (
  -0.00010799174582,
   0.00078642308446,
```

```
    -0.00261076151615,
     0.00649806802572,
    -0.01384614786056,
     0.02731224181000,
    -0.05475666574790,
     0.14572862668091,
     0.44870434767338,
    -0.08298074413630,
     0.03826155991708,
    -0.01954294245257,
     0.00962094760445,
    -0.00422447784966,
     0.00150611930873,
    -0.00034860279577); -- double

  -- Signals
  SIGNAL delay_pipeline                     : delay_pipeline_type(0 TO
max_flength-1) := (others => 0.0); -- double
  SIGNAL product                   : delay_pipeline_type(1 TO max_flength) :=
(others => 0.0); -- double
  SIGNAL sum                          : delay_pipeline_type(1 TO max_flength-1)
:= (others => 0.0); -- double

  SIGNAL output_register                    : real := 0.0; -- double


BEGIN

  -- Block Statements
  Delay_Pipeline_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      delay_pipeline(0 TO (max_flength-1)) <= (OTHERS =>
0.0000000000000000E+000);
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        delay_pipeline(0) <= filter_in;
        delay_pipeline(1 TO (max_flength-1)) <= delay_pipeline(0 TO
(max_flength-2));
      END IF;
    END IF;
  END PROCESS Delay_Pipeline_process;

  ----------------------------------------------------------------------------
------

  product_sum: for cnt in max_flength downto 0 generate
      product_process: if (cnt > 0) generate
            product(cnt) <= delay_pipeline(cnt-1) * coeff(cnt);
      end generate product_process;

      sum_first: if (cnt = 1) generate
            sum(cnt) <= product(cnt) + product(cnt+1);
      end generate sum_first;

      sum_process: if (cnt > 1 and cnt < max_flength) generate
            sum(cnt) <= sum(cnt-1) + product(cnt+1);
      end generate sum_process;
  end generate product_sum;
```

```
  -----------------------------------------------------------------------------
------

  Output_Register_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      output_register <= 0.0000000000000000E+000;
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        output_register <= sum(max_flength - 1);
      END IF;
    END IF;
  END PROCESS Output_Register_process;

  -- Assignment Statements
  filter_out <= output_register;

END rtl;
```

```
-- ----------------------------------------------------------------
-- Module: I/Qfir_filtbt
-- Original genertion by MATLAB(R) 7.0.4 and the Filter Design HDL Coder 1.2.
-- Edit and Update by Lt Jesse Somann: 2006-01-20
-- ----------------------------------------------------------------
-- HDL Code Generation Options:
--
-- TargetLanguage: VHDL
-- Name: I/Qfir_filtbt
-- TargetDirectory: C:\Documents and Settings\Owner\Desktop\AFIT
classes\Thesis\Matlab\hdlsrc
-- LoopUnrolling: On
-- InlineConfigurations: Off
-- SafeZeroConcat: Off
-- TestBenchName: I/Qfir_filtbt_tb
-- TestBenchStimulus: impulse step ramp chirp noise
--
-- Filter Settings:
--
-- Discrete-Time FIR Filter (real)
-- ------------------------------
-- Filter Structure  : Direct-Form FIR
-- Filter Length     : 16
-- ----------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;

ENTITY I/Qfir_filtbt IS

   GENERIC(   max_flength          :   natural := 16);

   PORT( clk                           :   IN    std_logic;
         clk_enable                    :   IN    std_logic;
         reset                         :   IN    std_logic;
         filter_in                     :   IN    real; -- double
         filter_out                    :   OUT   real  -- double
         );

END I/Qfir_filtbt;


----------------------------------------------------------------
--Module Architecture: I/Qfir_filtbt
----------------------------------------------------------------
ARCHITECTURE rtl OF I/Qfir_filtbt IS
  -- Local Functions
  -- Type Definitions

  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>) OF real; -- double
  TYPE coeff_pipeline_type IS ARRAY (NATURAL range <>) OF real; -- double

  -- Constants
  CONSTANT coeff                     : coeff_pipeline_type(1 TO max_flength) := (
  -0.00034860279577,
   0.00150611930873,
```

```
    -0.00422447784966,
     0.00962094760445,
    -0.01954294245257,
     0.03826155991708,
    -0.08298074413630,
     0.44870434767338,
     0.14572862668091,
    -0.05475666574790,
     0.02731224181000,
    -0.01384614786056,
     0.00649806802572,
    -0.00261076151615,
     0.00078642308446,
    -0.00010799174582); -- double

  -- Signals
  SIGNAL delay_pipeline                     : delay_pipeline_type(0 TO
max_flength-1) := (others => 0.0); -- double
  SIGNAL product                  : delay_pipeline_type(1 TO max_flength) :=
(others => 0.0); -- double
  SIGNAL sum                           : delay_pipeline_type(1 TO max_flength-1)
:= (others => 0.0); -- double

  SIGNAL output_register                    : real := 0.0; -- double


BEGIN

  -- Block Statements
  Delay_Pipeline_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      delay_pipeline(0 TO (max_flength-1)) <= (OTHERS =>
0.0000000000000000E+000);
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        delay_pipeline(0) <= filter_in;
        delay_pipeline(1 TO (max_flength-1)) <= delay_pipeline(0 TO
(max_flength-2));
      END IF;
    END IF;
  END PROCESS Delay_Pipeline_process;

  ----------------------------------------------------------------------------
------

  product_sum: for cnt in max_flength downto 0 generate
      product_process: if (cnt > 0) generate
            product(cnt) <= delay_pipeline(cnt-1) * coeff(cnt);
      end generate product_process;

      sum_first: if (cnt = 1) generate
            sum(cnt) <= product(cnt) + product(cnt+1);
      end generate sum_first;

      sum_process: if (cnt > 1 and cnt < max_flength) generate
            sum(cnt) <= sum(cnt-1) + product(cnt+1);
      end generate sum_process;
  end generate product_sum;

--  product_process : PROCESS (clk, reset)
```

```
--   variable counter : integer;
--   BEGIN
--     for counter in max_flength downto 1 loop
--             product(counter) <= delay_pipeline(counter-1) * coeff(counter);
--     end loop;
--   END PROCESS product_process;
--
--   sum_process : PROCESS (clk, reset)
--   variable counter : integer;
--   BEGIN
--     sum(1) <= product(1) + product(2);
--     for counter in 2 to (max_flength - 1) loop
--             sum(counter) <= sum(counter-1) + product(counter+1);
--     end loop;
--   END PROCESS sum_process;

  ----------------------------------------------------------------------
------

  Output_Register_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      output_register <= 0.0000000000000000E+000;
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        output_register <= sum(max_flength - 1);
      END IF;
    END IF;
  END PROCESS Output_Register_process;

  -- Assignment Statements
  filter_out <= output_register;

END rtl;
```

```
-----------------------------------------------------------
-- Module: I/Qoutput
-- Generated by Jesse P Somann
-- Started on 11 Oct 05
-- Finished on __
-- Purpose: To reform the output from 4 filters into I and Q data
--   streams
-- ---------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

ENTITY I/Qoutput IS
   PORT( clk                        :   IN    std_logic;
         clk_enable                 :   IN    std_logic;
         reset                      :   IN    std_logic;
         data_in_cnvlra             :   IN    real; -- double
         data_in_cnvlrb             :   IN    real; -- double
         data_in_cnvlia             :   IN    real; -- double
         data_in_cnvlib             :   IN    real; -- double
         dataI_out                  :   OUT   real; -- double
         dataQ_out          :   OUT   real
         );

END I/Qoutput;


-------------------------------------------------------------
--Module Architecture: testfir
-------------------------------------------------------------
ARCHITECTURE rtl OF I/Qoutput IS
  -- Local Functions

  SIGNAL cnvli_delayed                 : real := 0.0;

  SIGNAL output_registerI              : real := 0.0; -- double
  SIGNAL output_registerQ              : real := 0.0; -- double

  SIGNAL counter2                : std_logic;
  SIGNAL q_zero                         : std_logic;
  SIGNAL clk2                  : std_logic;

BEGIN

  Output_Register_process : PROCESS (clk, reset)
  BEGIN

    IF reset = '1' THEN
      output_registerI <= 0.0000000000000000E+000;
      output_registerQ <= 0.0000000000000000E+000;
      q_zero <= '0';
      counter2 <= '0';

    ELSIF clk'event AND clk = '1' THEN
```

```
        IF clk_enable = '1' AND q_zero = '0' THEN
         output_registerI <= data_in_cnvlrb;
         output_registerQ <= 0.0000000000000000E+000;
         cnvli_delayed <= data_in_cnvlib;
         counter2 <= '1';
         q_zero <= '1';

        ELSIF clk_enable = '1' THEN
          CASE counter2 is
                when '0' =>
                   output_registerI <= data_in_cnvlrb;
                   output_registerQ <= cnvli_delayed;
                   cnvli_delayed <= data_in_cnvlib;
                when '1' =>
                   output_registerI <= data_in_cnvlra;
                   output_registerQ <= cnvli_delayed;
                   cnvli_delayed <= data_in_cnvlia;
                when others =>
                   output_registerI <= 0.0;
                   output_registerQ <= 0.0;
          END CASE;

         IF counter2 = '0' THEN
           counter2 <= '1'; END IF;
         IF counter2 = '1' THEN
           counter2 <= '0'; END IF;

        END IF;
      END IF;
  END PROCESS Output_Register_process;

  -- Assignment Statements
  dataI_out <= output_registerI;
  dataQ_out <= output_registerQ;

END rtl;
```

147

```vhdl
-- --------------------------------------------------------------
-- Module: I/Q_sss_converter ('real' variables)
-- Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: To convert input data stream into I/Q data streams
-- --------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity I/Q _sss_converter is
    PORT( clk_in                          :   IN    std_logic;
        clk_x2_in                         :   IN    std_logic;
         clk_enable_in                    :   IN    std_logic;
         reset_in                         :   IN    std_logic;
         x_in                      :   IN    real; -- double
         It_out                           :   OUT   real; -- double
         Qt_out                           :   OUT   real  -- double
         );
end I/Q _sss_converter;

----------------------------------------------------------------
--Module Architecture: I/Q _sss_converter
----------------------------------------------------------------
architecture structure of I/Q _sss_converter is

component I/Qgen
   PORT( clk                       :   IN    std_logic;
         clk_enable                :   IN    std_logic;
         reset                     :   IN    std_logic;
         data_in                   :   IN    real; -- double
         dataI_out                 :   OUT   real;  -- double
         dataQ_out             :   OUT    real
         );
end component;

component I/Qfir_filtat
   PORT( clk                       :   IN    std_logic;
         clk_enable                :   IN    std_logic;
         reset                     :   IN    std_logic;
         filter_in                 :   IN    real; -- double
         filter_out                :   OUT   real  -- double
         );
end component;

component I/Qfir_filtbt
   PORT( clk                       :   IN    std_logic;
         clk_enable                :   IN    std_logic;
         reset                     :   IN    std_logic;
         filter_in                 :   IN    real; -- double
         filter_out                :   OUT   real  -- double
         );
end component;

component I/Qoutput
   PORT( clk                              :   IN    std_logic;
```

148

```
        clk_enable                          :   IN    std_logic;
        reset                               :   IN    std_logic;
        data_in_cnvlra                      :   IN    real; -- double
        data_in_cnvlrb                      :   IN    real; -- double
        data_in_cnvlia                      :   IN    real; -- double
        data_in_cnvlib                      :   IN    real; -- double
        dataI_out                           :   OUT   real; -- double
        dataQ_out                   :   OUT   real
        );
end component;

signal xr, xi, cnvlra, cnvlrb, cnvlia, cnvlib: real;

begin
        g1: I/Qgen port map (data_in=>x_in, clk=>clk_in,
clk_enable=>clk_enable_in, reset=>reset_in, dataI_out=>xr, dataQ_out=>xi);
        g2: I/Qfir_filtat port map (filter_in=>xr, clk=>clk_x2_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlra);
        g3: I/Qfir_filtat port map (filter_in=>xi, clk=>clk_x2_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlia);
        g4: I/Qfir_filtbt port map (filter_in=>xr, clk=>clk_x2_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlrb);
        g5: I/Qfir_filtbt port map (filter_in=>xi, clk=>clk_x2_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlib);
        g6: I/Qoutput port map (data_in_cnvlra=>cnvlra, data_in_cnvlrb=>cnvlrb,
data_in_cnvlia=>cnvlia, data_in_cnvlib=>cnvlib, clk=>clk_in,
clk_enable=>clk_enable_in, reset=>reset_in, dataI_out=>It_out,
dataQ_out=>Qt_out);

end structure;
```

```
-- ---------------------------------------------------------------
-- Module: I/Qgen_dig
-- Generated by Jesse P Somann
-- Last edited: 30 Nov 06
-- Purpose: to split the input data into a real (I) and
--  imaginary (Q) stream initiative to be convolved
-- ---------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
--use ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;

ENTITY I/Qgen_dig IS

   GENERIC( bit_width                    :    natural := 6);

   PORT( clk                             :    IN    std_logic;
         clk_enable                      :    IN    std_logic;
         reset                           :    IN    std_logic;
         data_in                         :    IN    std_logic_vector (bit_width-
1  downto 0);
         dataI_out                       :    OUT   std_logic_vector (bit_width-
1  downto 0);
         dataQ_out                 :    OUT   std_logic_vector (bit_width-1
downto 0)
         );
END I/Qgen_dig;


-- ---------------------------------------------------------------
--Module Architecture: I/Qgen_dig
-- ---------------------------------------------------------------
ARCHITECTURE rtl OF I/Qgen_dig IS
 -- Local Functions

  -- Signals
  SIGNAL output_registerI                  : std_logic_vector (bit_width-1
downto 0) := "000000";
  SIGNAL output_registerQ                  : std_logic_vector (bit_width-1
downto 0) := "000000";

  SIGNAL counter4                    : std_logic_vector (1 downto 0);
  SIGNAL inverted                    : std_logic_vector (bit_width-1  downto 0)
:= "000000";

BEGIN

  Output_Register_process : PROCESS (clk, reset)
  BEGIN

    IF reset = '1' THEN
      output_registerI <= "000000";
      output_registerQ <= "000000";
      counter4 <= "00";

    ELSIF clk'event AND clk = '1' THEN
```

150

```vhdl
      IF clk_enable = '1' THEN

        CASE counter4 is
              when "00" =>
                output_registerI <= data_in;
              when "01" =>
                output_registerQ <= data_in;
              when "10" =>
                output_registerI <= (not(data_in)) + '1';
              when "11" =>
                output_registerQ <= (not(data_in)) + '1';
              when others =>
                output_registerI <= "000000";
                output_registerQ <= "000000";
        END CASE;
        counter4 <= counter4 + "01";

      END IF;
    END IF;
  END PROCESS Output_Register_process;

  -- Assignment Statements
  dataI_out <= output_registerI;
  dataQ_out <= output_registerQ;

END rtl;
```

```
-- ----------------------------------------------------------------
-- Module: I/Qfir_filtat_dig
-- Original: Generated by MATLAB(R) 7.0.4 and the Filter Design HDL Coder 1.2.
-- Re-Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: Low pass filter for steaming data input.
-- ----------------------------------------------------------------
-- Filter Settings:
--
-- Discrete-Time FIR Filter (real)
-- -------------------------------
-- Filter Structure  : Direct-Form FIR
-- Filter Length     : 16
-- Stable            : Yes
-- Linear Phase      : Yes (Type 2)
-- ----------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;
--USE ieee.std_logic_signed.all;

ENTITY I/Qfir_filtat_dig IS

   GENERIC( max_flength                  :   natural := 16;
            bit_width_in         :   natural := 6;
            bit_width_coeff      :   natural := 6;
            bit_width_out        :   natural := 16);

   PORT( clk                             :   IN    std_logic;
         clk_enable                      :   IN    std_logic;
         reset                           :   IN    std_logic;
         filter_in                       :   IN    std_logic_vector
(bit_width_in-1 downto 0);
         filter_out                      :   OUT   std_logic_vector
(bit_width_out-1 downto 0)
         );

END I/Qfir_filtat_dig;


-- ----------------------------------------------------------------
--Module Architecture: I/Qfir_filtat_dig
-- ----------------------------------------------------------------
ARCHITECTURE rtl OF I/Qfir_filtat_dig IS
  -- Local Functions
  -- Type Definitions

  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_in-1 downto 0); -- double
  TYPE coeff_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_coeff-1 downto 0); -- double
  TYPE product_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_in + bit_width_coeff-1 downto 0); -- double
  TYPE sum_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
((bit_width_out-1) downto 0); -- double

  -- Constants
  CONSTANT coeff                        : coeff_pipeline_type(1 TO max_flength) := (
```

```vhdl
    "000000",
    "000000",
    "000000",
    "000000",
    "000000",
    "000001",
    "111110",
    "000101",
    "001110",
    "111101",
    "000001",
    "111111",
    "000000",
    "000000",
    "000000",
    "000000");

  -- Signals
  SIGNAL delay_pipeline                      : delay_pipeline_type(0 TO
max_flength-1) := (others => "000000"); -- double
  SIGNAL product                    : product_pipeline_type(1 TO max_flength) :=
(others => "000000000000"); -- double
  SIGNAL sum                              : sum_pipeline_type(1 TO max_flength-1)
:= (others => "0000000000000000"); -- double

  SIGNAL output_register                    : std_logic_vector (bit_width_out-1
downto 0) := "0000000000000000";

BEGIN
  -- Block Statements
  Delay_Pipeline_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      delay_pipeline(0 TO (max_flength-1)) <= (OTHERS => "000000");
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        delay_pipeline(0) <= filter_in;
        delay_pipeline(1 TO (max_flength-1)) <= delay_pipeline(0 TO
(max_flength-2));
      END IF;
    END IF;
  END PROCESS Delay_Pipeline_process;
  -----------------------------------------------------------------------------
------
  product_sum: for cnt in max_flength downto 0 generate

      product_process: if (cnt > 0) generate
                  P : PROCESS (delay_pipeline(cnt-1))
                          variable a_int:          SIGNED ((bit_width_in-1)
downto 0);
                          variable b_int:          SIGNED ((bit_width_coeff-
1) downto 0);
                          variable pdt_int:        SIGNED ((bit_width_in +
bit_width_coeff-1) downto 0);
                      BEGIN
                          a_int := SIGNED (delay_pipeline(cnt-1));
                          b_int := SIGNED (coeff(cnt));
                          pdt_int := a_int * b_int;
                          product(cnt) <= STD_LOGIC_VECTOR(pdt_int);
                  END PROCESS P;
      end generate product_process;
```

```vhdl
      sum_first: if (cnt = 1) generate
                  S1 : PROCESS (product(cnt),product(cnt+1))
                        variable a_int:          SIGNED ((bit_width_out-1)
downto 0);
                        variable b_int:          SIGNED ((bit_width_out-1)
downto 0);
                        variable pdt_int:        SIGNED ((bit_width_out-1)
downto 0);
                     BEGIN
                        a_int := "0000" & (SIGNED (product(cnt)));
                        b_int := "0000" & (SIGNED (product(cnt+1)));
                        for pad in 0 to (bit_width_out - bit_width_in -
bit_width_coeff - 1) loop
                              a_int(bit_width_in + bit_width_coeff + pad) :=
a_int(bit_width_in + bit_width_coeff-1);
                              b_int(bit_width_in + bit_width_coeff + pad) :=
b_int(bit_width_in + bit_width_coeff-1);
                        end loop;
                        pdt_int := (a_int + b_int);
                        sum(cnt) <= STD_LOGIC_VECTOR(pdt_int);
                     END PROCESS S1;
      end generate sum_first;

      sum_process: if (cnt > 1 and cnt < max_flength) generate
                  S : PROCESS (sum(cnt-1),product(cnt+1))
                        variable a_int:          SIGNED ((bit_width_out-1)
downto 0);
                        variable b_int:          SIGNED ((bit_width_out-1)
downto 0);
                        variable pdt_int:        SIGNED ((bit_width_out-1)
downto 0);
                     BEGIN
                        a_int := (SIGNED (sum(cnt-1)));
                        b_int := "0000" & (SIGNED (product(cnt+1)));
                        for pad in 0 to (bit_width_out - bit_width_in -
bit_width_coeff - 1) loop
                              b_int(bit_width_in + bit_width_coeff + pad) :=
b_int(bit_width_in + bit_width_coeff-1);
                        end loop;
                        pdt_int := (a_int + b_int);
                        sum(cnt) <= STD_LOGIC_VECTOR(pdt_int);
                     END PROCESS S;
      end generate sum_process;
  end generate product_sum;
  ------------------------------------------------------------------------
------
  Output_Register_process : PROCESS (clk, reset)

            variable full:                    SIGNED ((bit_width_out - 1)
downto 0);
--          variable trunk:          SIGNED ((bit_width_out-1) downto 0);

      BEGIN
        IF reset = '1' THEN
          output_register <= "0000000000000000";

        ELSIF clk'event AND clk = '1' THEN
          IF clk_enable = '1' THEN
            full := SIGNED(sum(max_flength - 1));
```

```
--          trunk := full((bit_width_in + bit_width_coeff + max_flength - 1)
downto (bit_width_in + bit_width_coeff + max_flength - 1 - bit_width_out + 1));
            output_register <= STD_LOGIC_VECTOR(full);
          END IF;
        END IF;

      END PROCESS Output_Register_process;

  -- Assignment Statements
  filter_out <= output_register;

END rtl;
```

```
-- ----------------------------------------------------------------
-- Module: I/Qoutput_dig
-- Generated by Jesse P Somann
-- Last edited: 30 Nov 06
-- Purpose: To reform the output from 4 filters into I and Q data
--   streams
-- ----------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
--USE IEEE.numeric_std.ALL;
--use ieee.std_logic_arith.all;
--USE ieee.std_logic_signed.all;
--use ieee.std_logic_unsigned.all;

ENTITY I/Qoutput_dig IS

   GENERIC( bit_width                    :    natural := 16);

   PORT( clk                      :    IN    std_logic;
         clk_enable               :    IN    std_logic;
         reset                    :    IN    std_logic;
         data_in_cnvlra           :    IN    std_logic_vector (bit_width-
1 downto 0);
         data_in_cnvlrb           :    IN    std_logic_vector (bit_width-
1 downto 0);
         data_in_cnvlia           :    IN    std_logic_vector (bit_width-
1 downto 0);
         data_in_cnvlib           :    IN    std_logic_vector (bit_width-
1 downto 0);
         dataI_out                :    OUT   std_logic_vector (bit_width-
1 downto 0);
         dataQ_out                :    OUT   std_logic_vector (bit_width-1
downto 0)
         );

END I/Qoutput_dig;

-- ----------------------------------------------------------------
--Module Architecture: I/Qoutput_dig
-- ----------------------------------------------------------------
ARCHITECTURE rtl OF I/Qoutput_dig IS
  -- Local Functions

  SIGNAL cnvli_delayed               : std_logic_vector (bit_width-1
downto 0) := "0000000000000000";
  SIGNAL output_registerI            : std_logic_vector (bit_width-1
downto 0) := "0000000000000000";
  SIGNAL output_registerQ            : std_logic_vector (bit_width-1
downto 0) := "0000000000000000";

  SIGNAL counter2              : std_logic;
  SIGNAL q_zero                      : std_logic;
  SIGNAL clk2                  : std_logic;

BEGIN

  Output_Register_process : PROCESS (clk, reset)
  BEGIN
```

156

```vhdl
    IF reset = '1' THEN
      output_registerI <= "0000000000000000";
      output_registerQ <= "0000000000000000";
      q_zero <= '0';
      counter2 <= '0';

    ELSIF clk'event AND clk = '1' THEN

      IF clk_enable = '1' AND q_zero = '0' THEN
       output_registerI <= data_in_cnvlrb;
       output_registerQ <= "0000000000000000";
       cnvli_delayed <= data_in_cnvlib;
       counter2 <= '1';
       q_zero <= '1';

      ELSIF clk_enable = '1' THEN
        CASE counter2 is
             when '0' =>
               output_registerI <= data_in_cnvlrb;
               output_registerQ <= cnvli_delayed;
               cnvli_delayed <= data_in_cnvlib;
             when '1' =>
               output_registerI <= data_in_cnvlra;
               output_registerQ <= cnvli_delayed;
               cnvli_delayed <= data_in_cnvlia;
             when others =>
               output_registerI <= "0000000000000000";
               output_registerQ <= "0000000000000000";
        END CASE;

      IF counter2 = '0' THEN
        counter2 <= '1'; END IF;
      IF counter2 = '1' THEN
        counter2 <= '0'; END IF;

      END IF;
    END IF;
  END PROCESS Output_Register_process;

  -- Assignment Statements
  dataI_out <= output_registerI;
  dataQ_out <= output_registerQ;

END rtl;
```

157

Appendix M: tsui_sss_converter_dig (VHDL)

```
-- ------------------------------------------------------------
-- Module: tsui_sss_converter_dig
-- Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: To convert input data stream into I/Q data streams
-- ------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;

entity tsui_sss_converter_dig is
   GENERIC( bit_width_in          :   natural := 6;
            bit_width_out         :   natural := 16);

    PORT( clk_in                              :   IN    std_logic;
         clk_x2_in                            :   IN    std_logic;
          clk_enable_in                       :   IN    std_logic;
          reset_in                            :   IN    std_logic;
          x_in                      :   IN    std_logic_vector (bit_width_in-
1 downto 0);
          It_out                          :   OUT   std_logic_vector
(bit_width_out-1 downto 0);
          Qt_out                          :   OUT   std_logic_vector
(bit_width_out-1 downto 0)
          );
end tsui_sss_converter_dig;
------------------------------------------------------------
--Module Architecture: tsui_sss_converter_dig
------------------------------------------------------------
architecture structure of tsui_sss_converter_dig is

component I/Qgen_dig
   PORT( clk                        :   IN    std_logic;
         clk_enable                 :   IN    std_logic;
         reset                      :   IN    std_logic;
         data_in                    :   IN    std_logic_vector
(bit_width_in-1 downto 0);
         dataI_out                  :   OUT   std_logic_vector
(bit_width_in-1 downto 0);
         dataQ_out              :   OUT   std_logic_vector (bit_width_in-1
downto 0)
         );
end component;

component I/Qfir_filtat_dig
   PORT( clk                        :   IN    std_logic;
         clk_enable                 :   IN    std_logic;
         reset                      :   IN    std_logic;
         filter_in                  :   IN    std_logic_vector
(bit_width_in-1 downto 0);
         filter_out                 :   OUT   std_logic_vector
(bit_width_out-1 downto 0)
         );
end component;

component I/Qfir_filtbt_dig
   PORT( clk                                :   IN    std_logic;
```

```vhdl
        clk_enable                              :   IN    std_logic;
        reset                                   :   IN    std_logic;
        filter_in                               :   IN    std_logic_vector
(bit_width_in-1 downto 0);
        filter_out                              :   OUT   std_logic_vector
(bit_width_out-1 downto 0)
        );
end component;

component I/Qoutput_dig
   PORT( clk                                    :   IN    std_logic;
        clk_enable                              :   IN    std_logic;
        reset                                   :   IN    std_logic;
        data_in_cnvlra                          :   IN    std_logic_vector
(bit_width_out-1 downto 0);
        data_in_cnvlrb                          :   IN    std_logic_vector
(bit_width_out-1 downto 0);
        data_in_cnvlia                          :   IN    std_logic_vector
(bit_width_out-1 downto 0);
        data_in_cnvlib                          :   IN    std_logic_vector
(bit_width_out-1 downto 0);
        dataI_out                               :   OUT   std_logic_vector
(bit_width_out-1 downto 0);
        dataQ_out                    :   OUT   std_logic_vector (bit_width_out-1
downto 0)
        );
end component;

signal xr, xi:                                  std_logic_vector (bit_width_in-1
downto 0);
signal cnvlra, cnvlrb, cnvlia, cnvlib:          std_logic_vector (bit_width_out-1
downto 0);

begin
        g1: I/Qgen_dig port map (data_in=>x_in, clk=>clk_in,
clk_enable=>clk_enable_in, reset=>reset_in, dataI_out=>xr, dataQ_out=>xi);
        g2: I/Qfir_filtat_dig port map (filter_in=>xr, clk=>clk_x2_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlra);
        g3: I/Qfir_filtat_dig port map (filter_in=>xi, clk=>clk_x2_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlia);
        g4: I/Qfir_filtbt_dig port map (filter_in=>xr, clk=>clk_x2_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlrb);
        g5: I/Qfir_filtbt_dig port map (filter_in=>xi, clk=>clk_x2_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlib);
        g6: I/Qoutput_dig port map (data_in_cnvlra=>cnvlra,
data_in_cnvlrb=>cnvlrb, data_in_cnvlia=>cnvlia, data_in_cnvlib=>cnvlib,
clk=>clk_in, clk_enable=>clk_enable_in, reset=>reset_in, dataI_out=>It_out,
dataQ_out=>Qt_out);

end structure;
```

```vhdl
-- ---------------------------------------------------------------
-- Design: sss_pipeline (varied)
-- To generate: tsui_sss_converter_struct_pipeline_dig
-- Original:
-- Re-Generated by Jesse P Somann
-- Last edited: 06 Feb 06
-- Purpose: I/Q data conversion
-- ---------------------------------------------------------------


-- ---------------------------------------------------------------
-- ---------------------------------------------------------------
-- Register
-- ---------------------------------------------------------------
-- ---------------------------------------------------------------
----http://www.csee.umbc.edu/help/VHDL/samples/reg_g.vhdl

library ieee;
use ieee.std_logic_1164.all;
entity reg is
  generic(left   : natural);      -- top bit
  port  (clk   : in  std_logic;
          input : in  std_logic_vector (left downto 0);
          output : out std_logic_vector (left downto 0) );
end entity reg;

architecture behavior of reg is
begin  -- behavior
  reg: process(clk)
       begin
         if clk='1' then               -- rising edge
           output <= input;
         end if;
       end process reg;
end architecture behavior;

-- ---------------------------------------------------------------
-- ---------------------------------------------------------------
-- Unsigned Adder
-- ---------------------------------------------------------------
-- ---------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;


ENTITY add_unsigned IS
      GENERIC( bit_width_in                 :   natural;
              bit_width_out                 :   natural);
      PORT (
              a:            IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
              b:            IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
              result:       OUT STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0)
      );
END add_unsigned;

ARCHITECTURE rtl OF add_unsigned IS
```

```
BEGIN

   S : PROCESS (a,b)
                        variable a_int:
       STD_LOGIC_VECTOR((bit_width_in-1) downto 0);
                        variable b_int:
       STD_LOGIC_VECTOR(bit_width_in-1) downto 0);
                        variable pdt_int:
       STD_LOGIC_VECTOR((bit_width_out-1) downto 0);
                     BEGIN
                        --use a for loop using the variable
                        a_int := (a);
                        b_int := (b);
                        pdt_int := (a_int + b_int);
                        result <= STD_LOGIC_VECTOR(pdt_int);
                     END PROCESS S;

END rtl;

-- ----------------------------------------------------------------
-- ----------------------------------------------------------------
-- Lookahead Adder for first signed addition
-- ----------------------------------------------------------------
-- ----------------------------------------------------------------


-- ----------------------------------------------------------------
-- Module: add_signed_lookahead_first (behavioral)
-- Original: http://www.altera.com/support/examples/vhdl/v_cl_addr.html
-- Re-Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: Signed carry lookahead adder, adjustable bit-widths
-- ----------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY add_signed_lookahead_first IS
       GENERIC( bit_width_in                   :   natural;
                bit_width_out                  :   natural;
                bit_string             :   std_logic_vector;
                bit_string_cii                 :   std_logic_vector;
                pad_string             :   std_logic_vector;
                pad_width              :   natural);
       PORT (
               a:            IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
               b:            IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
               result:       OUT STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0)
       );

END add_signed_lookahead_first;

ARCHITECTURE behavioral OF add_signed_lookahead_first IS

SIGNAL    a_pad              :      STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
       := bit_string;
SIGNAL    b_pad              :      STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
       := bit_string;
SIGNAL    h_sum              :      STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
       := bit_string;
SIGNAL    carry_generate     :      STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
       := bit_string;
```

```
SIGNAL    carry_propagate   :    STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
      := bit_string;
SIGNAL    carry_in_internal :    STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 1)
      := bit_string_cii;

BEGIN

    PAD:PROCESS (a,b)
            variable a_int:            STD_LOGIC_VECTOR((bit_width_out - 1)
downto 0);
            variable b_int:            STD_LOGIC_VECTOR((bit_width_out - 1)
downto 0);
      BEGIN
            a_int := pad_string & (a);
            b_int := pad_string & (b);
            for pad in 0 to (pad_width-1) loop
                  a_int((bit_width_in) + pad) := a_int(bit_width_in - 1);
                  b_int((bit_width_in) + pad) := b_int(bit_width_in - 1);
            end loop;
            a_pad <= STD_LOGIC_VECTOR(a_int);
            b_pad <= STD_LOGIC_VECTOR(b_int);
    END PROCESS PAD;

    h_sum <= a_pad XOR b_pad;
    carry_generate <= a_pad AND b_pad;
    carry_propagate <= a_pad OR b_pad;

    PROCESS (carry_generate,carry_propagate,carry_in_internal)
          BEGIN
          carry_in_internal(1) <= carry_generate(0);
            inst: FOR i IN 1 TO (bit_width_out-2) LOOP
                carry_in_internal(i+1) <= carry_generate(i) OR
(carry_propagate(i) AND carry_in_internal(i));
                END LOOP;
--        result(bit_width_out-1) <= carry_generate(bit_width_in-1) OR
(carry_propagate(bit_width_in-1) AND carry_in_internal(bit_width_in-1));
    END PROCESS;

    result(0) <= h_sum(0);
    result((bit_width_out-1) DOWNTO 1) <= h_sum((bit_width_out-1) DOWNTO 1) XOR
carry_in_internal((bit_width_out-1) DOWNTO 1);
END behavioral;


-- -------------------------------------------------------------
-- -------------------------------------------------------------
-- Lookahead Adder for subsequent signed additions
-- -------------------------------------------------------------
-- -------------------------------------------------------------


-- -------------------------------------------------------------
-- Module: add_signed_lookahead (behavioral)
-- Original: http://www.altera.com/support/examples/vhdl/v_cl_addr.html
-- Re-Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: Signed carry lookahead adder, adjustable bit-widths
-- -------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY add_signed_lookahead IS
```

```vhdl
        GENERIC( bit_width_in                    :    natural;
                bit_width_out                    :    natural;
                bit_string                  :    std_logic_vector;
                bit_string_cii                    :    std_logic_vector);
        PORT (
                a:              IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
                b:              IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
                result:         OUT STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0)
        );

END add_signed_lookahead;

ARCHITECTURE behavioral OF add_signed_lookahead IS

SIGNAL    a_pad            :    STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
        := bit_string;
SIGNAL    b_pad            :    STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
        := bit_string;
SIGNAL    h_sum            :    STD_LOGIC_VECTOR((bit_width_in-1) DOWNTO 0)
        := bit_string;
SIGNAL    carry_generate   :    STD_LOGIC_VECTOR((bit_width_in-1) DOWNTO 0)
        := bit_string;
SIGNAL    carry_propagate  :    STD_LOGIC_VECTOR((bit_width_in-1) DOWNTO 0)
        := bit_string;
SIGNAL    carry_in_internal :   STD_LOGIC_VECTOR((bit_width_in-1) DOWNTO 1)
        := bit_string_cii;

BEGIN

    PAD:PROCESS (a,b)
            variable a_int:             STD_LOGIC_VECTOR((bit_width_out - 1)
downto 0);
            variable b_int:             STD_LOGIC_VECTOR((bit_width_out - 1)
downto 0);
        BEGIN
            a_int := (a);
            b_int := (b);
            a_int(bit_width_in - 1) := a_int(bit_width_in - 2);
            b_int(bit_width_in - 1) := b_int(bit_width_in - 2);
            a_pad <= STD_LOGIC_VECTOR(a_int);
            b_pad <= STD_LOGIC_VECTOR(b_int);
    END PROCESS PAD;

    h_sum <= a_pad XOR b_pad;
    carry_generate <= a_pad AND b_pad;
    carry_propagate <= a_pad OR b_pad;

    PROCESS (carry_generate,carry_propagate,carry_in_internal)
        BEGIN
        carry_in_internal(1) <= carry_generate(0);
          inst: FOR i IN 1 TO (bit_width_in-2) LOOP
                carry_in_internal(i+1) <= carry_generate(i) OR
(carry_propagate(i) AND carry_in_internal(i));
                END LOOP;
--      result(bit_width_out-1) <= carry_generate(bit_width_in-1) OR
(carry_propagate(bit_width_in-1) AND carry_in_internal(bit_width_in-1));
    END PROCESS;

    result(0) <= h_sum(0);
    result((bit_width_out-1) DOWNTO 1) <= h_sum((bit_width_out-1) DOWNTO 1) XOR
carry_in_internal((bit_width_out-1) DOWNTO 1);
```

163

```
END behavioral;




-- -------------------------------------------------------------
-- -------------------------------------------------------------
-- Multiplier using custom method for signed 2's compliment (pipelined)
-- -------------------------------------------------------------
-- -------------------------------------------------------------


-- -------------------------------------------------------------
-- Module: mult_signed_custom_pipeline (behavioral)
-- Original: http://www.altera.com/support/examples/vhdl/v_cl_addr.html
-- Re-Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: Signed carry lookahead adder, adjustable bit-widths
-- -------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY mult_signed_custom_pipeline IS
      GENERIC( bit_width_in1                  :   natural;  --in1 <= in2
              bit_width_in2                  :   natural;
              bit_width_out                  :   natural;
              bit_width_out_fin         :   natural;
              bit_string_in1                  :   std_logic_vector;
              bit_string_in2                  :   std_logic_vector;
              bit_string_out                  :   std_logic_vector;
              pad_string                 :   std_logic_vector);
      PORT (
              clk:                   IN  std_logic;
              clk_enable:   in  std_logic;
              reset:                 in  std_logic;
              a:               IN STD_LOGIC_VECTOR ((bit_width_in1-1) DOWNTO 0);
              b:               IN STD_LOGIC_VECTOR ((bit_width_in2-1) DOWNTO 0);
              result:        OUT STD_LOGIC_VECTOR ((bit_width_out_fin-1) DOWNTO
0));

END mult_signed_custom_pipeline;

ARCHITECTURE behavioral OF mult_signed_custom_pipeline IS

--components
 component add_unsigned
   GENERIC( bit_width_in                  :   natural;
           bit_width_out                  :   natural);
   PORT( a:            IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
        b:            IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
        result:       OUT STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0));
 end component;

 component reg
  generic(left   : natural);      -- top bit
  port   (clk    : in  std_logic;
          input : in  std_logic_vector (left downto 0);
          output : out std_logic_vector (left downto 0) );
 end component;

--types
```

```vhdl
TYPE sums_type IS ARRAY (NATURAL range <>) OF std_logic_vector (bit_width_out-1
downto 0);
TYPE delay_type IS ARRAY (NATURAL range <>) OF std_logic_vector (bit_width_in1-
1 downto 0);


--signals
SIGNAL    delay_pipeline_a   :    delay_type(0 TO bit_width_in1-1) := (others
=> bit_string_in1);
SIGNAL    delay_pipeline_b   :    delay_type(0 TO bit_width_in1-1) := (others
=> bit_string_in1);
SIGNAL         sums             :    sums_type(0 TO bit_width_in1-1) := (others
=> bit_string_in & bit_string_in);
SIGNAL         added                   :    sums_type(0 TO bit_width_in1-1) :=
(others => bit_string_in & bit_string_in);
SIGNAL         added_reg       :    sums_type(0 TO bit_width_in1-1) := (others
=> bit_string_in & bit_string_in);
SIGNAL    a_pad              :    STD_LOGIC_VECTOR((bit_width_in1-1) DOWNTO 0)
     := bit_string_in1;
SIGNAL    b_pad              :    STD_LOGIC_VECTOR((bit_width_in1-1) DOWNTO 0)
     := bit_string_in1;
SIGNAL    out_reg            :    STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
     := bit_string_in1 & bit_string_in1;

BEGIN

  PAD:PROCESS (a,b)
       variable a_int:                   STD_LOGIC_VECTOR((bit_width_in1 - 1)
downto 0);
       variable b_int:              STD_LOGIC_VECTOR((bit_width_in1 - 1) downto
0);
       BEGIN
         if (bit_width_in1 - bit_width_in2) > 0 then
                a_int := (a);
                b_int := pad_string & (b);
                for pad in 0 to ((bit_width_in1 - bit_width_in2) - 1) loop
                      b_int((bit_width_in2) + pad) := b_int(bit_width_in2 - 1);
                end loop;
         else
                a_int:= (a);
                b_int:= (b);
         end if;
         a_pad <= STD_LOGIC_VECTOR(a_int);
         b_pad <= STD_LOGIC_VECTOR(b_int);
  END PROCESS PAD;

 ------------------------------------------------------------------------------
-------------------
 Delay_Pipeline_process : PROCESS (clk, reset, a_pad, b_pad)
  BEGIN
    IF reset = '1' THEN
      delay_pipeline_a(0 TO (bit_width_in1-1)) <= (OTHERS => bit_string_in1);
      delay_pipeline_b(0 TO (bit_width_in1-1)) <= (OTHERS => bit_string_in1);
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        delay_pipeline_a(0) <= a_pad;
        delay_pipeline_b(0) <= b_pad;
        delay_pipeline_a(1 TO (bit_width_in1-1)) <= delay_pipeline_a(0 TO
(bit_width_in1-2));
        delay_pipeline_b(1 TO (bit_width_in1-1)) <= delay_pipeline_b(0 TO
(bit_width_in1-2));
      END IF;
```

165

```
    END IF;
  END PROCESS Delay_Pipeline_process;

 -------------------------------------------------------------------------------
-------------------
  intermediates: for cnt in (bit_width_in1 - 1) downto 0 generate

       first: if (cnt = 0) generate
              fi:FOR i IN 0 TO (bit_width_in1-2) generate
               sums(cnt)(i) <= delay_pipeline_a(cnt)(cnt) AND
delay_pipeline_b(cnt)(i);
              END generate fi;
              sums(cnt)(bit_width_in1-1) <= delay_pipeline_a(cnt)(cnt) NAND
delay_pipeline_b(cnt)(bit_width_in1-1);
              sums(cnt)(bit_width_in1) <= '1';
              P: add_unsigned generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out)
                           port map(a=>added(cnt), b=>sums(cnt),
result=>added_reg(cnt));
              Pr: reg generic map (left=>(bit_width_out-1))
                      port map(clk=>clk, input=>added_reg(cnt),
output=>added(cnt+1));
       end generate first;

       middle: if (cnt > 0 and cnt < (bit_width_in1 - 1)) generate
              mi:FOR i IN cnt TO (bit_width_in1 - 2 + cnt) generate
               sums(cnt)(i) <= delay_pipeline_a(cnt)(cnt) AND
delay_pipeline_b(cnt)(i-cnt);
              END generate mi;
              sums(cnt)(bit_width_in1 - 1 + cnt) <= delay_pipeline_a(cnt)(cnt)
NAND delay_pipeline_b(cnt)(bit_width_in1-1);
              P: add_unsigned generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out)
                           port map(a=>added(cnt), b=>sums(cnt),
result=>added_reg(cnt));
              Pr: reg generic map (left=>(bit_width_out-1))
                      port map(clk=>clk, input=>added_reg(cnt),
output=>added(cnt+1));
       end generate middle;

       last: if (cnt = (bit_width_in1 - 1)) generate
              la:FOR i IN cnt TO (bit_width_in1 - 2 + cnt) generate
               sums(cnt)(i) <= delay_pipeline_a(cnt)(cnt) NAND
delay_pipeline_b(cnt)(i-cnt);
              END generate la;
              sums(cnt)(bit_width_in1 - 1 + cnt) <= delay_pipeline_a(cnt)(cnt)
AND delay_pipeline_b(cnt)(bit_width_in1-1);
              sums(cnt)(bit_width_out-1) <= '1';
              P: add_unsigned generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out)
                           port map(a=>added(cnt), b=>sums(cnt),
result=>added_reg(cnt));
       end generate last;

  end generate intermediates;

----------------------------------------------------------------------------
-------------------
  Output_Register_process : PROCESS (clk, reset, added_reg)
       BEGIN
          IF reset = '1' THEN
```

166

```
            out_reg <= bit_string_in1 & bit_string_in1;
          ELSIF clk'event AND clk = '1' THEN
            IF clk_enable = '1' THEN
              out_reg <= added_reg(bit_width_in1 - 1);
            END IF;
          END IF;
  END PROCESS Output_Register_process;
        result <= out_reg((bit_width_out_fin-1) DOWNTO 0);

END behavioral;


-- --------------------------------------------------------------
-- --------------------------------------------------------------
-- Input Data Splitter for incoming data stream
-- --------------------------------------------------------------
-- --------------------------------------------------------------


-- --------------------------------------------------------------
-- Module: I/Qgen_dig
-- Generated by Jesse P Somann
-- Last edited: 30 Nov 06
-- Purpose: to split the input data into a real (I) and
--  imaginary (Q) stream initiative to be convolved
-- --------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
--use ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;

ENTITY I/Qgen_dig IS

   GENERIC( bit_width                    :   natural;
            bit_string                   :   std_logic_vector);

   PORT( clk                             :   IN    std_logic;
         clk_enable                      :   IN    std_logic;
         reset                           :   IN    std_logic;
         data_in                         :   IN    std_logic_vector (bit_width-
1  downto 0);
         dataI_out                       :   OUT   std_logic_vector (bit_width-
1  downto 0);
         dataQ_out                :   OUT   std_logic_vector (bit_width-1
downto 0)
         );
END I/Qgen_dig;


----------------------------------------------------------------
--Module Architecture: I/Qgen_dig
----------------------------------------------------------------
ARCHITECTURE rtl OF I/Qgen_dig IS
 -- Local Functions

  -- Signals
  SIGNAL output_registerI                : std_logic_vector (bit_width-1
downto 0) := bit_string;
  SIGNAL output_registerQ                : std_logic_vector (bit_width-1
downto 0) := bit_string;
```

167

```vhdl
  SIGNAL counter4                    : std_logic_vector (1 downto 0);
  SIGNAL inverted                    : std_logic_vector (bit_width-1  downto 0)
:= bit_string;

BEGIN

  Output_Register_process : PROCESS (clk, reset)
  BEGIN

    IF reset = '1' THEN
      output_registerI <= bit_string;
      output_registerQ <= bit_string;
      counter4 <= "00";

    ELSIF clk'event AND clk = '1' THEN

      IF clk_enable = '1' THEN

        CASE counter4 is
            when "00" =>
              output_registerI <= data_in;
            when "01" =>
              output_registerQ <= data_in;
            when "10" =>
              output_registerI <= (not(data_in)) + '1';
            when "11" =>
              output_registerQ <= (not(data_in)) + '1';
            when others =>
              output_registerI <= bit_string;
              output_registerQ <= bit_string;
        END CASE;
        counter4 <= counter4 + "01";

      END IF;
    END IF;
  END PROCESS Output_Register_process;

  -- Assignment Statements
  dataI_out <= output_registerI;
  dataQ_out <= output_registerQ;

END rtl;




-- ------------------------------------------------------------
-- ------------------------------------------------------------
-- Filter 'a', low pass, pipelined
-- ------------------------------------------------------------
-- ------------------------------------------------------------


-- ------------------------------------------------------------
-- Module: I/Qfir_filtat_dig_struct2_pipeline
-- Original: Generated by MATLAB(R) 7.0.4 and the Filter Design HDL Coder 1.2.
-- Re-Generated by Jesse P Somann
-- Last edited: 06 Feb 06
-- Purpose: Low pass filter for steaming data input.
-- ------------------------------------------------------------
-- Filter Settings:
--
-- Discrete-Time FIR Filter (real)
```

```
-- ------------------------------
-- Filter Structure  : Direct-Form FIR
-- Filter Length     : 16
-- Stable            : Yes
-- Linear Phase      : Yes (Type 2)
-- ----------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;
--USE ieee.std_logic_signed.all;


ENTITY I/Qfir_filtat_dig_struct2_pipeline IS

   GENERIC( max_flength                 :   natural;
            levels              :    natural;
            bit_width_in        :    natural;
            bit_width_coeff     :    natural;
            bit_width_mult_pad           :   natural;
            bit_width_out       :    natural;
            bit_string_in       :    std_logic_vector;
            bit_string_coeff             :   std_logic_vector;
            bit_string_in_coeff          :   std_logic_vector;
            bit_string_out      :    std_logic_vector;
            bit_string_cii      :    std_logic_vector;
            pad_string                   :   std_logic_vector;
            pad_string_add      :    std_logic_vector);

   PORT( clk                            :   IN    std_logic;
         clk_enable                     :   IN    std_logic;
         reset                          :   IN    std_logic;
         filter_in                      :   IN    std_logic_vector
(bit_width_in-1 downto 0);
         filter_out                     :   OUT   std_logic_vector
(bit_width_out-1 downto 0)
         );

END I/Qfir_filtat_dig_struct2_pipeline;



-----------------------------------------------------------------
--Module Architecture: I/Qfir_filtat_dig_struct2_pipeline
-----------------------------------------------------------------
ARCHITECTURE rtl OF I/Qfir_filtat_dig_struct2_pipeline IS
  -- Local Functions

  -- Components
  component add_signed_lookahead_first
   GENERIC( bit_width_in                :    natural;
              bit_width_out                     :    natural;
              bit_string            :    std_logic_vector;
              bit_string_cii                :    std_logic_vector;
              pad_string            :    std_logic_vector;
              pad_width             :    natural);
   PORT( a:          IN STD_LOGIC_VECTOR ((bit_width_in - 1) DOWNTO 0);
        b:          IN STD_LOGIC_VECTOR ((bit_width_in - 1) DOWNTO 0);
       result:     OUT STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0));
   end component;

  component add_signed_lookahead
   GENERIC( bit_width_in                :    natural;
```

```
                bit_width_out                          :    natural;
                bit_string                      :    std_logic_vector;
                bit_string_cii                       :    std_logic_vector);
   PORT( a:            IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
        b:            IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
        result:      OUT STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0));
  end component;


  component mult_signed_custom_pipeline
   generic( bit_width_in1                   :    natural;  --in1 <= in2
          bit_width_in2                  :    natural;
          bit_width_out                  :    natural;
          bit_width_out_fin                   :    natural;
          bit_string_in1                 :    std_logic_vector;
          bit_string_in2                 :    std_logic_vector;
          bit_string_out                 :    std_logic_vector;
          pad_string                          :    std_logic_vector);
   PORT( clk:            IN  std_logic;
        clk_enable:       in  std_logic;
        reset:            in  std_logic;
        a:            IN STD_LOGIC_VECTOR ((bit_width_in1 - 1) DOWNTO 0);
        b:            IN STD_LOGIC_VECTOR ((bit_width_in2 - 1) DOWNTO 0);
        result:      OUT STD_LOGIC_VECTOR ((bit_width_out_fin - 1) DOWNTO 0));
  end component;


  component reg
   generic(left   : natural);       -- top bit
   port   (clk    : in  std_logic;
           input  : in  std_logic_vector (left downto 0);
           output : out std_logic_vector (left downto 0) );
  end component;


  -- Type Definitions
  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_in-1 downto 0); -- double
  TYPE coeff_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_coeff-1 downto 0); -- double
  TYPE product_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_in + bit_width_coeff-1 downto 0); -- double
  TYPE sum_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
((bit_width_out-1) downto 0); -- double

  -- Constants
  CONSTANT coeff                     : coeff_pipeline_type(1 TO max_flength) := (
   --cofat--

  -- Signals
  SIGNAL delay_pipeline                       : delay_pipeline_type(0 TO
max_flength-1) := (others => bit_string_in);
  SIGNAL product                    : product_pipeline_type(1 TO max_flength) :=
(others => bit_string_in_coeff);
  SIGNAL sum                         : sum_pipeline_type(1 TO max_flength-1)
:= (others => bit_string_out);
  SIGNAL product_reg                         : product_pipeline_type(1 TO
max_flength) := (others => bit_string_in_coeff);
  SIGNAL sum_reg                     : sum_pipeline_type(1 TO max_flength-1) :=
(others => bit_string_out);

  SIGNAL output_register                  : std_logic_vector (bit_width_out-1
downto 0) := bit_string_out;
```

```vhdl
    SIGNAL clk2                      : std_logic := '0';

BEGIN

  -- Block Statements
  clk2_process : PROCESS (clk, reset)
  variable switch:               STD_LOGIC := '1';
  BEGIN
   IF clk'event AND clk = '1' AND switch = '1' THEN
      IF clk_enable = '1' THEN
        clk2 <= NOT clk2;
        switch := '0';
      END IF;
    END IF;
    IF clk'event AND clk = '1' AND switch = '0' THEN
      IF clk_enable = '1' THEN
        switch := '1';
      END IF;
    END IF;
  END PROCESS clk2_process;

  ------------------------------------------------------------------------------
------

  Delay_Pipeline_process : PROCESS (clk2, reset)
  BEGIN
    IF reset = '1' THEN
      delay_pipeline(0 TO (max_flength-1)) <= (OTHERS => bit_string_in);
    ELSIF clk2'event AND clk2 = '1' THEN
      IF clk_enable = '1' THEN
        delay_pipeline(0) <= filter_in;
        delay_pipeline(1 TO (max_flength-1)) <= delay_pipeline(0 TO
(max_flength-2));
      END IF;
    END IF;
  END PROCESS Delay_Pipeline_process;

  ------------------------------------------------------------------------------
------

  product_sum: for cnt in max_flength downto 0 generate


      product_process: if (cnt > 0) generate
            greater1: IF (bit_width_in >= bit_width_coeff) generate
                P: mult_signed_custom_pipeline generic map
(bit_width_in1=>bit_width_in, bit_width_in2=>bit_width_coeff,
bit_width_out=>(bit_width_in + bit_width_coeff + bit_width_mult_pad),
bit_width_out_fin=>(bit_width_in + bit_width_coeff),
bit_string_in1=>bit_string_in, bit_string_in2=>bit_string_coeff,
bit_string_out=>bit_string_in_coeff, pad_string=>pad_string)
                                              port map(clk=>clk,
clk_enable=>clk_enable, reset=>reset, a=>delay_pipeline(cnt-1), b=>coeff(cnt),
result=>product(cnt));
            END generate greater1;
            greater2: IF (bit_width_in < bit_width_coeff) generate
                P: mult_signed_custom_pipeline generic map
(bit_width_in1=>bit_width_coeff, bit_width_in2=>bit_width_in,
bit_width_out=>(bit_width_in + bit_width_coeff + bit_width_mult_pad),
bit_width_out_fin=>(bit_width_in + bit_width_coeff),
```

171

```vhdl
bit_string_in1=>bit_string_coeff, bit_string_in2=>bit_string_in,
bit_string_out=>bit_string_in_coeff, pad_string=>pad_string)
                                           port map(clk=>clk,
clk_enable=>clk_enable, reset=>reset, a=>coeff(cnt), b=>delay_pipeline(cnt-1),
result=>product(cnt));
                END generate greater2;
                    Pr: reg generic map (left=>(bit_width_in + bit_width_coeff -
1))
                            port map(clk=>clk, input=>product(cnt),
output=>product_reg(cnt));
        end generate product_process;



        sum_tree1: if (cnt > 0 and cnt <= max_flength/2 and levels>=1) generate
                P: add_signed_lookahead_first generic map
(bit_width_in=>(bit_width_in + bit_width_coeff), bit_width_out=>bit_width_out,
bit_string=>bit_string_out, bit_string_cii=>bit_string_cii,
pad_string=>pad_string_add, pad_width=>levels)
                                          port map(a=>product(cnt*2),
b=>product(cnt*2 - 1), result=>sum(cnt));
                Pr: reg generic map (left=>(bit_width_out -1))
                        port map(clk=>clk, input=>sum(cnt), output=>sum_reg(cnt));
        end generate sum_tree1;


        sum_tree2: if (cnt > 0 and cnt <= max_flength/4 and levels>=2) generate
                P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                          port map(a=>sum(cnt*2), b=>sum(cnt*2 - 1),
result=>sum(cnt + max_flength/2));
                Pr: reg generic map (left=>(bit_width_out -1))
                        port map(clk=>clk, input=>sum(cnt + max_flength/2),
output=>sum_reg(cnt + max_flength/2));
        end generate sum_tree2;


        sum_tree3: if (cnt > 0 and cnt <= max_flength/8 and levels>=3) generate
                P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                          port map(a=>sum(max_flength/2 + cnt*2),
b=>sum(max_flength/2 + cnt*2 - 1), result=>sum(cnt + (max_flength/4)*3));
                Pr: reg generic map (left=>(bit_width_out -1))
                        port map(clk=>clk, input=>sum(cnt + (max_flength/4)*3),
output=>sum_reg(cnt + (max_flength/4)*3));
        end generate sum_tree3;


        sum_tree4: if (cnt > 0 and cnt <= max_flength/16 and levels>=4) generate
                P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                          port map(a=>sum((max_flength/4)*3 + cnt*2),
b=>sum((max_flength/4)*3 + cnt*2 - 1), result=>sum(cnt + (max_flength/8)*7));
                Pr: reg generic map (left=>(bit_width_out -1))
                        port map(clk=>clk, input=>sum(cnt + (max_flength/8)*7),
output=>sum_reg(cnt + (max_flength/8)*7));
        end generate sum_tree4;
```

```vhdl
        sum_tree5: if (cnt > 0 and cnt <= max_flength/32 and levels>=5) generate
              P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                   port map(a=>sum((max_flength/8)*7 + cnt*2),
b=>sum((max_flength/8)*7 + cnt*2 - 1), result=>sum(cnt + (max_flength/16)*15));
              Pr: reg generic map (left=>(bit_width_out -1))
                      port map(clk=>clk, input=>sum(cnt + (max_flength/16)*15),
output=>sum_reg(cnt + (max_flength/16)*15));
        end generate sum_tree5;


        sum_tree6: if (cnt > 0 and cnt <= max_flength/64 and levels>=6) generate
              P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                   port map(a=>sum((max_flength/16)*15 + cnt*2),
b=>sum((max_flength/16)*15 + cnt*2 - 1), result=>sum(cnt +
(max_flength/32)*31));
              Pr: reg generic map (left=>(bit_width_out -1))
                      port map(clk=>clk, input=>sum(cnt + (max_flength/32)*31),
output=>sum_reg(cnt + (max_flength/32)*31));
        end generate sum_tree6;


        sum_tree7: if (cnt > 0 and cnt <= max_flength/128 and levels>=7) generate
              P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                   port map(a=>sum((max_flength/32)*31 + cnt*2),
b=>sum((max_flength/32)*31 + cnt*2 - 1), result=>sum(cnt +
(max_flength/64)*63));
              Pr: reg generic map (left=>(bit_width_out -1))
                      port map(clk=>clk, input=>sum(cnt + (max_flength/64)*63),
output=>sum_reg(cnt + (max_flength/64)*63));
        end generate sum_tree7;


  end generate product_sum;

  ----------------------------------------------------------------------------
------

  Output_Register_process : PROCESS (clk2, reset)

              variable full:                     STD_LOGIC_VECTOR ((bit_width_out
- 1) downto 0);
--            variable trunk:          STD_LOGIC_VECTOR ((bit_width_out - 1)
downto 0);

        BEGIN
          IF reset = '1' THEN
            output_register <= bit_string_out;

          ELSIF clk2'event AND clk2 = '1' THEN
            IF clk_enable = '1' THEN
              full := STD_LOGIC_VECTOR(sum(max_flength - 1));
--            trunk := full((bit_width_in + bit_width_coeff + max_flength - 1)
downto (bit_width_in + bit_width_coeff + max_flength - 1 - bit_width_out + 1));
              output_register <= STD_LOGIC_VECTOR(full);
            END IF;
```

173

```
        END IF;

    END PROCESS Output_Register_process;

  -- Assignment Statements
  filter_out <= output_register;

END rtl;


-- -----------------------------------------------------------
-- -----------------------------------------------------------
-- Filter 'b', low pass, pipelined
-- -----------------------------------------------------------
-- -----------------------------------------------------------


-- -----------------------------------------------------------
-- Module: I/Qfir_filtbt_dig_struct2_pipeline
-- Original: Generated by MATLAB(R) 7.0.4 and the Filter Design HDL Coder 1.2.
-- Re-Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: Low pass filter for steaming data input.
-- -----------------------------------------------------------
-- Filter Settings:
--
-- Discrete-Time FIR Filter (real)
-- -------------------------------
-- Filter Structure  : Direct-Form FIR
-- Filter Length     : 16
-- Stable            : Yes
-- Linear Phase      : Yes (Type 2)
-- -----------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;
--USE ieee.std_logic_signed.all;

ENTITY I/Qfir_filtbt_dig_struct2_pipeline IS

   GENERIC( max_flength                :   natural;
            levels               :   natural;
            bit_width_in         :   natural;
            bit_width_coeff      :   natural;
            bit_width_mult_pad          :   natural;
            bit_width_out        :   natural;
            bit_string_in        :   std_logic_vector;
            bit_string_coeff            :   std_logic_vector;
            bit_string_in_coeff         :   std_logic_vector;
            bit_string_out       :   std_logic_vector;
            bit_string_cii       :   std_logic_vector;
            pad_string                  :   std_logic_vector;
            pad_string_add       :   std_logic_vector);

   PORT( clk                            :   IN    std_logic;
         clk_enable                     :   IN    std_logic;
         reset                          :   IN    std_logic;
         filter_in                      :   IN    std_logic_vector
(bit_width_in-1 downto 0);
         filter_out                     :   OUT   std_logic_vector
(bit_width_out-1 downto 0)
```

```
         );

END I/Qfir_filtbt_dig_struct2_pipeline;


----------------------------------------------------------------
--Module Architecture: I/Qfir_filtbt_dig_struct2_pipeline
----------------------------------------------------------------
ARCHITECTURE rtl OF I/Qfir_filtbt_dig_struct2_pipeline IS
  -- Local Functions

  -- Components
  component add_signed_lookahead_first
   GENERIC( bit_width_in                  :   natural;
                 bit_width_out                   :    natural;
                 bit_string                  :   std_logic_vector;
                 bit_string_cii                    :    std_logic_vector;
                 pad_string                  :   std_logic_vector;
                 pad_width                  :   natural);
   PORT( a:           IN STD_LOGIC_VECTOR ((bit_width_in - 1) DOWNTO 0);
         b:           IN STD_LOGIC_VECTOR ((bit_width_in - 1) DOWNTO 0);
         result:      OUT STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0));
  end component;


  component add_signed_lookahead
   GENERIC( bit_width_in                  :   natural;
                 bit_width_out                   :    natural;
                 bit_string                  :   std_logic_vector;
                 bit_string_cii                    :    std_logic_vector);
   PORT( a:           IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
         b:           IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
         result:      OUT STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0));
  end component;


  component mult_signed_custom_pipeline
   generic( bit_width_in1                 :   natural;  --in1 <= in2
            bit_width_in2                 :   natural;
            bit_width_out                 :   natural;
            bit_width_out_fin              :    natural;
            bit_string_in1                :   std_logic_vector;
            bit_string_in2                :   std_logic_vector;
            bit_string_out                :   std_logic_vector;
            pad_string                  :    std_logic_vector);
   PORT( clk:              IN  std_logic;
         clk_enable:       in  std_logic;
         reset:            in  std_logic;
         a:           IN STD_LOGIC_VECTOR ((bit_width_in1 - 1) DOWNTO 0);
         b:           IN STD_LOGIC_VECTOR ((bit_width_in2 - 1) DOWNTO 0);
         result:      OUT STD_LOGIC_VECTOR ((bit_width_out_fin - 1) DOWNTO 0));
  end component;

  component reg
   generic(left   : natural);       -- top bit
   port   (clk    : in  std_logic;
            input  : in  std_logic_vector (left downto 0);
            output : out std_logic_vector (left downto 0) );
  end component;

  -- Type Definitions
  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_in-1 downto 0); -- double
```

```vhdl
  TYPE coeff_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_coeff-1 downto 0); -- double
  TYPE product_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_in + bit_width_coeff-1 downto 0); -- double
  TYPE sum_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
((bit_width_out-1) downto 0); -- double

  -- Constants
  CONSTANT coeff                    : coeff_pipeline_type(1 TO max_flength) := (
   --cofbt--

  -- Signals
  SIGNAL delay_pipeline             : delay_pipeline_type(0 TO
max_flength-1) := (others => bit_string_in);
  SIGNAL product                    : product_pipeline_type(1 TO max_flength) :=
(others => bit_string_in_coeff);
  SIGNAL sum                        : sum_pipeline_type(1 TO max_flength-1)
:= (others => bit_string_out);
  SIGNAL product_reg                : product_pipeline_type(1 TO
max_flength) := (others => bit_string_in_coeff);
  SIGNAL sum_reg                    : sum_pipeline_type(1 TO max_flength-1) :=
(others => bit_string_out);

  SIGNAL output_register            : std_logic_vector (bit_width_out-1
downto 0) := bit_string_out;

  SIGNAL clk2              : std_logic := '0';


BEGIN

  -- Block Statements
  clk2_process : PROCESS (clk, reset)
  variable switch:            STD_LOGIC := '1';
  BEGIN
   IF clk'event AND clk = '1' AND switch = '1' THEN
      IF clk_enable = '1' THEN
        clk2 <= NOT clk2;
        switch := '0';
      END IF;
    END IF;
    IF clk'event AND clk = '1' AND switch = '0' THEN
      IF clk_enable = '1' THEN
        switch := '1';
      END IF;
    END IF;
  END PROCESS clk2_process;

  ------------------------------------------------------------------------------
------

  Delay_Pipeline_process : PROCESS (clk2, reset)
  BEGIN
    IF reset = '1' THEN
      delay_pipeline(0 TO (max_flength-1)) <= (OTHERS => bit_string_in);
    ELSIF clk2'event AND clk2 = '1' THEN
      IF clk_enable = '1' THEN
        delay_pipeline(0) <= filter_in;
        delay_pipeline(1 TO (max_flength-1)) <= delay_pipeline(0 TO
(max_flength-2));
      END IF;
```

176

```
   END IF;
 END PROCESS Delay_Pipeline_process;

 --------------------------------------------------------------------------------
------

 product_sum: for cnt in max_flength downto 0 generate


       product_process: if (cnt > 0) generate
              greater1: IF (bit_width_in >= bit_width_coeff) generate
                  P: mult_signed_custom_pipeline generic map
(bit_width_in1=>bit_width_in, bit_width_in2=>bit_width_coeff,
bit_width_out=>(bit_width_in + bit_width_coeff + bit_width_mult_pad),
bit_width_out_fin=>(bit_width_in + bit_width_coeff),
bit_string_in1=>bit_string_in, bit_string_in2=>bit_string_coeff,
bit_string_out=>bit_string_in_coeff, pad_string=>pad_string)
                                        port map(clk=>clk,
clk_enable=>clk_enable, reset=>reset, a=>delay_pipeline(cnt-1), b=>coeff(cnt),
result=>product(cnt));
              END generate greater1;
              greater2: IF (bit_width_in < bit_width_coeff) generate
                  P: mult_signed_custom_pipeline generic map
(bit_width_in1=>bit_width_coeff, bit_width_in2=>bit_width_in,
bit_width_out=>(bit_width_in + bit_width_coeff + bit_width_mult_pad),
bit_width_out_fin=>(bit_width_in + bit_width_coeff),
bit_string_in1=>bit_string_coeff, bit_string_in2=>bit_string_in,
bit_string_out=>bit_string_in_coeff, pad_string=>pad_string)
                                        port map(clk=>clk,
clk_enable=>clk_enable, reset=>reset, a=>coeff(cnt), b=>delay_pipeline(cnt-1),
result=>product(cnt));
              END generate greater2;
                  Pr: reg generic map (left=>(bit_width_in + bit_width_coeff -
1))
                         port map(clk=>clk, input=>product(cnt),
output=>product_reg(cnt));
       end generate product_process;


       sum_tree1: if (cnt > 0 and cnt <= max_flength/2 and levels>=1) generate
              P: add_signed_lookahead_first generic map
(bit_width_in=>(bit_width_in + bit_width_coeff), bit_width_out=>bit_width_out,
bit_string=>bit_string_out, bit_string_cii=>bit_string_cii,
pad_string=>pad_string_add, pad_width=>levels)
                                        port map(a=>product(cnt*2),
b=>product(cnt*2 - 1), result=>sum(cnt));
              Pr: reg generic map (left=>(bit_width_out -1))
                     port map(clk=>clk, input=>sum(cnt), output=>sum_reg(cnt));
       end generate sum_tree1;


       sum_tree2: if (cnt > 0 and cnt <= max_flength/4 and levels>=2) generate
              P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                        port map(a=>sum(cnt*2), b=>sum(cnt*2 - 1),
result=>sum(cnt + max_flength/2));
              Pr: reg generic map (left=>(bit_width_out -1))
                     port map(clk=>clk, input=>sum(cnt + max_flength/2),
output=>sum_reg(cnt + max_flength/2));
       end generate sum_tree2;
```

```
      sum_tree3: if (cnt > 0 and cnt <= max_flength/8 and levels>=3) generate
             P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                     port map(a=>sum(max_flength/2 + cnt*2),
b=>sum(max_flength/2 + cnt*2 - 1), result=>sum(cnt + (max_flength/4)*3));
             Pr: reg generic map (left=>(bit_width_out -1))
                     port map(clk=>clk, input=>sum(cnt + (max_flength/4)*3),
output=>sum_reg(cnt + (max_flength/4)*3));
      end generate sum_tree3;


      sum_tree4: if (cnt > 0 and cnt <= max_flength/16 and levels>=4) generate
             P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                     port map(a=>sum((max_flength/4)*3 + cnt*2),
b=>sum((max_flength/4)*3 + cnt*2 - 1), result=>sum(cnt + (max_flength/8)*7));
             Pr: reg generic map (left=>(bit_width_out -1))
                     port map(clk=>clk, input=>sum(cnt + (max_flength/8)*7),
output=>sum_reg(cnt + (max_flength/8)*7));
      end generate sum_tree4;


      sum_tree5: if (cnt > 0 and cnt <= max_flength/32 and levels>=5) generate
             P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                     port map(a=>sum((max_flength/8)*7 + cnt*2),
b=>sum((max_flength/8)*7 + cnt*2 - 1), result=>sum(cnt + (max_flength/16)*15));
             Pr: reg generic map (left=>(bit_width_out -1))
                     port map(clk=>clk, input=>sum(cnt + (max_flength/16)*15),
output=>sum_reg(cnt + (max_flength/16)*15));
      end generate sum_tree5;


      sum_tree6: if (cnt > 0 and cnt <= max_flength/64 and levels>=6) generate
             P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                     port map(a=>sum((max_flength/16)*15 + cnt*2),
b=>sum((max_flength/16)*15 + cnt*2 - 1), result=>sum(cnt +
(max_flength/32)*31));
             Pr: reg generic map (left=>(bit_width_out -1))
                     port map(clk=>clk, input=>sum(cnt + (max_flength/32)*31),
output=>sum_reg(cnt + (max_flength/32)*31));
      end generate sum_tree6;


      sum_tree7: if (cnt > 0 and cnt <= max_flength/128 and levels>=7) generate
             P: add_signed_lookahead generic map (bit_width_in=>bit_width_out,
bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii)
                                     port map(a=>sum((max_flength/32)*31 + cnt*2),
b=>sum((max_flength/32)*31 + cnt*2 - 1), result=>sum(cnt +
(max_flength/64)*63));
             Pr: reg generic map (left=>(bit_width_out -1))
                     port map(clk=>clk, input=>sum(cnt + (max_flength/64)*63),
output=>sum_reg(cnt + (max_flength/64)*63));
```

```vhdl
    end generate sum_tree7;


  end generate product_sum;

  -------------------------------------------------------------------------------
------

  Output_Register_process : PROCESS (clk2, reset)

            variable full:                 STD_LOGIC_VECTOR ((bit_width_out
- 1) downto 0);
--          variable trunk:         STD_LOGIC_VECTOR ((bit_width_out - 1)
downto 0);

      BEGIN
        IF reset = '1' THEN
          output_register <= bit_string_out;

        ELSIF clk2'event AND clk2 = '1' THEN
          IF clk_enable = '1' THEN
            full := STD_LOGIC_VECTOR(sum(max_flength - 1));
--          trunk := full((bit_width_in + bit_width_coeff + max_flength - 1)
downto (bit_width_in + bit_width_coeff + max_flength - 1 - bit_width_out + 1));
            output_register <= STD_LOGIC_VECTOR(full);
          END IF;
        END IF;

      END PROCESS Output_Register_process;

  -- Assignment Statements
  filter_out <= output_register;

END rtl;


-- -------------------------------------------------------------
-- -------------------------------------------------------------
-- Recombination Unit for making final I and Q channels
-- -------------------------------------------------------------
-- -------------------------------------------------------------


-- -------------------------------------------------------------
-- Module: I/Qoutput_dig
-- Generated by Jesse P Somann
-- Last edited: 30 Nov 06
-- Purpose: To reform the output from 4 filters into I and Q data
--   streams
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
--USE IEEE.numeric_std.ALL;
--use ieee.std_logic_arith.all;
--USE ieee.std_logic_signed.all;
--use ieee.std_logic_unsigned.all;

ENTITY I/Qoutput_dig IS

   GENERIC( bit_width                 :    natural;
         bit_string                   :    std_logic_vector);
```

```
   PORT( clk                                    :    IN     std_logic;
         clk_enable                             :    IN     std_logic;
         reset                                  :    IN     std_logic;
         data_in_cnvlra                         :    IN     std_logic_vector (bit_width-
1 downto 0);
         data_in_cnvlrb                         :    IN     std_logic_vector (bit_width-
1 downto 0);
         data_in_cnvlia                         :    IN     std_logic_vector (bit_width-
1 downto 0);
         data_in_cnvlib                         :    IN     std_logic_vector (bit_width-
1 downto 0);
         dataI_out                              :    OUT    std_logic_vector (bit_width-
1 downto 0);
         dataQ_out                     :    OUT    std_logic_vector (bit_width-1
downto 0)
         );

END I/Qoutput_dig;


-----------------------------------------------------------------
--Module Architecture: I/Qoutput_dig
-----------------------------------------------------------------
ARCHITECTURE rtl OF I/Qoutput_dig IS
  -- Local Functions

  SIGNAL cnvli_delayed                          : std_logic_vector (bit_width-1
downto 0) := bit_string;
  SIGNAL output_registerI                       : std_logic_vector (bit_width-1
downto 0) := bit_string;
  SIGNAL output_registerQ                       : std_logic_vector (bit_width-1
downto 0) := bit_string;

  SIGNAL counter2                   : std_logic;
  SIGNAL q_zero                                 : std_logic;
  SIGNAL clk2                       : std_logic;

BEGIN

  Output_Register_process : PROCESS (clk, reset)
  BEGIN

    IF reset = '1' THEN
      output_registerI <= bit_string;
      output_registerQ <= bit_string;
      q_zero <= '0';
      counter2 <= '0';

    ELSIF clk'event AND clk = '1' THEN

      IF clk_enable = '1' AND q_zero = '0' THEN
       output_registerI <= data_in_cnvlrb;
       output_registerQ <= bit_string;
       cnvli_delayed <= data_in_cnvlib;
       counter2 <= '1';
       q_zero <= '1';

      ELSIF clk_enable = '1' THEN
        CASE counter2 is
             when '0' =>
               output_registerI <= data_in_cnvlrb;
               output_registerQ <= cnvli_delayed;
```

180

```
                cnvli_delayed <= data_in_cnvlib;
             when '1' =>
                output_registerI <= data_in_cnvlra;
                output_registerQ <= cnvli_delayed;
                cnvli_delayed <= data_in_cnvlia;
             when others =>
                output_registerI <= bit_string;
                output_registerQ <= bit_string;
      END CASE;

      IF counter2 = '0' THEN
        counter2 <= '1'; END IF;
      IF counter2 = '1' THEN
        counter2 <= '0'; END IF;

     END IF;
   END IF;
  END PROCESS Output_Register_process;

  -- Assignment Statements
  dataI_out <= output_registerI;
  dataQ_out <= output_registerQ;

END rtl;




-- --------------------------------------------------------------
-- --------------------------------------------------------------
-- Toplevel Structural Setup
-- --------------------------------------------------------------
-- --------------------------------------------------------------


-- --------------------------------------------------------------
-- Module: tsui_sss_converter_struct2
-- Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: To convert input data stream into I/Q data streams
-- --------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity tsui_sss_converter_struct2_pipeline_dig is
      GENERIC( max_flength          :    natural := --mfl--;
            levels                  :    natural := --lvs--;
            bit_width_in            :    natural := --bwi--;
            bit_width_coeff         :    natural := --bwc--;
            bit_width_mult_pad          :    natural := --bwmp--;
            bit_width_out           :    natural := --bwo--;
            bit_string_in           :    std_logic_vector (--bsin-- downto 0) := -
-bsi--; --"000000"
            bit_string_coeff            :    std_logic_vector (--bscn-- downto
0) := --bsc--;
            bit_string_in_coeff         :    std_logic_vector (--bsicn-- downto
0) := --bsic--;
            bit_string_out          :    std_logic_vector (--bson-- downto 0) := -
-bso--;
            bit_string_cii          :    std_logic_vector (--ciin-- downto 1) := -
-cii--;
               --psm--     --pad_string_mult
```

```vhdl
        pad_string_add                : std_logic_vector (--psan-- downto 0) := -
-psa--);
    PORT( clk_in                                : IN    std_logic;
        clk_enable_in                     : IN    std_logic;
        reset_in                          : IN    std_logic;
        x_in                          : IN    std_logic_vector (bit_width_in-
1 downto 0);
        It_out                            : OUT    std_logic_vector
(bit_width_out-1 downto 0);
        Qt_out                            : OUT    std_logic_vector
(bit_width_out-1 downto 0));
end tsui_sss_converter_struct2_pipeline_dig;

----------------------------------------------------------------
--Module Architecture: tsui_sss_converter_struct2_pipeline_dig
----------------------------------------------------------------
architecture structure of tsui_sss_converter_struct2_pipeline_dig is

component I/Qgen_dig
    GENERIC( bit_width                     : natural;
            bit_string                     : std_logic_vector);
    PORT( clk                          : IN    std_logic;
        clk_enable                     : IN    std_logic;
        reset                          : IN    std_logic;
        data_in                        : IN    std_logic_vector
(bit_width_in-1 downto 0);
        dataI_out                      : OUT    std_logic_vector
(bit_width_in-1 downto 0);
        dataQ_out                  : OUT    std_logic_vector (bit_width_in-1
downto 0)
        );
end component;

component I/Qfir_filtat_dig_struct2_pipeline
    GENERIC( max_flength                   : natural;
            levels                 : natural;
            bit_width_in           : natural;
            bit_width_coeff        : natural;
            bit_width_mult_pad          : natural;
            bit_width_out          : natural;
            bit_string_in          : std_logic_vector;
            bit_string_coeff           : std_logic_vector;
            bit_string_in_coeff        : std_logic_vector;
            bit_string_out         : std_logic_vector;
            bit_string_cii         : std_logic_vector;
            pad_string                 : std_logic_vector;
            pad_string_add         : std_logic_vector);
    PORT( clk                          : IN    std_logic;
        clk_enable                     : IN    std_logic;
        reset                          : IN    std_logic;
        filter_in                      : IN    std_logic_vector
(bit_width_in-1 downto 0);
        filter_out                         : OUT    std_logic_vector
(bit_width_out-1 downto 0)
        );
end component;

component I/Qfir_filtbt_dig_struct2_pipeline
    GENERIC( max_flength                   : natural;
            levels                 : natural;
            bit_width_in           : natural;
```

182

```vhdl
            bit_width_coeff             :   natural;
            bit_width_mult_pad                  :       natural;
            bit_width_out               :   natural;
            bit_string_in               :   std_logic_vector;
            bit_string_coeff                :   std_logic_vector;
            bit_string_in_coeff             :   std_logic_vector;
            bit_string_out              :   std_logic_vector;
            bit_string_cii              :   std_logic_vector;
            pad_string                      :       std_logic_vector;
            pad_string_add              :   std_logic_vector);
    PORT( clk                                       :   IN      std_logic;
        clk_enable                          :   IN      std_logic;
        reset                               :   IN      std_logic;
        filter_in                           :   IN      std_logic_vector
(bit_width_in-1 downto 0);
        filter_out                          :   OUT     std_logic_vector
(bit_width_out-1 downto 0)
        );
end component;

component I/Qoutput_dig
    GENERIC( bit_width                      :   natural;
            bit_string                      :   std_logic_vector);
    PORT( clk                                   :   IN      std_logic;
        clk_enable                          :   IN      std_logic;
        reset                               :   IN      std_logic;
        data_in_cnvlra                      :   IN      std_logic_vector
(bit_width_out-1 downto 0);
        data_in_cnvlrb                      :   IN      std_logic_vector
(bit_width_out-1 downto 0);
        data_in_cnvlia                      :   IN      std_logic_vector
(bit_width_out-1 downto 0);
        data_in_cnvlib                      :   IN      std_logic_vector
(bit_width_out-1 downto 0);
        dataI_out                           :   OUT     std_logic_vector
(bit_width_out-1 downto 0);
        dataQ_out                       :   OUT   std_logic_vector (bit_width_out-1
downto 0)
        );
end component;

signal xr, xi:                                  std_logic_vector (bit_width_in-1
downto 0);
signal cnvlra, cnvlrb, cnvlia, cnvlib:      std_logic_vector (bit_width_out-1
downto 0);

begin

    g1: I/Qgen_dig generic map (bit_width=>bit_width_in,
bit_string=>bit_string_in)
                    port map (data_in=>x_in, clk=>clk_in,
clk_enable=>clk_enable_in, reset=>reset_in, dataI_out=>xr, dataQ_out=>xi);
    g2: I/Qfir_filtat_dig_struct2_pipeline generic map
(max_flength=>max_flength, levels=>levels, bit_width_in=>bit_width_in,
bit_width_coeff=>bit_width_coeff, bit_width_mult_pad=>bit_width_mult_pad,
bit_width_out=>bit_width_out, bit_string_in=>bit_string_in,
bit_string_coeff=>bit_string_coeff, bit_string_in_coeff=>bit_string_in_coeff,
bit_string_out=>bit_string_out, bit_string_cii=>bit_string_cii,
pad_string=>pad_string, pad_string_add=>pad_string_add)
                                        port map (filter_in=>xr, clk=>clk_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlra);
```

```
        g3: I/Qfir_filtat_dig_struct2_pipeline generic map
(max_flength=>max_flength, levels=>levels, bit_width_in=>bit_width_in,
bit_width_coeff=>bit_width_coeff, bit_width_mult_pad=>bit_width_mult_pad,
bit_width_out=>bit_width_out, bit_string_in=>bit_string_in,
bit_string_coeff=>bit_string_coeff, bit_string_in_coeff=>bit_string_in_coeff,
bit_string_out=>bit_string_out, bit_string_cii=>bit_string_cii,
pad_string=>pad_string, pad_string_add=>pad_string_add)
                                      port map (filter_in=>xi, clk=>clk_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlia);
        g4: I/Qfir_filtbt_dig_struct2_pipeline generic map
(max_flength=>max_flength, levels=>levels, bit_width_in=>bit_width_in,
bit_width_coeff=>bit_width_coeff, bit_width_mult_pad=>bit_width_mult_pad,
bit_width_out=>bit_width_out, bit_string_in=>bit_string_in,
bit_string_coeff=>bit_string_coeff, bit_string_in_coeff=>bit_string_in_coeff,
bit_string_out=>bit_string_out, bit_string_cii=>bit_string_cii,
pad_string=>pad_string, pad_string_add=>pad_string_add)
                                      port map (filter_in=>xr, clk=>clk_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlrb);
        g5: I/Qfir_filtbt_dig_struct2_pipeline generic map
(max_flength=>max_flength, levels=>levels, bit_width_in=>bit_width_in,
bit_width_coeff=>bit_width_coeff, bit_width_mult_pad=>bit_width_mult_pad,
bit_width_out=>bit_width_out, bit_string_in=>bit_string_in,
bit_string_coeff=>bit_string_coeff, bit_string_in_coeff=>bit_string_in_coeff,
bit_string_out=>bit_string_out, bit_string_cii=>bit_string_cii,
pad_string=>pad_string, pad_string_add=>pad_string_add)
                                      port map (filter_in=>xi, clk=>clk_in,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlib);
        g6: I/Qoutput_dig generic map (bit_width=>bit_width_out,
bit_string=>bit_string_out)
                      port map (data_in_cnvlra=>cnvlra, data_in_cnvlrb=>cnvlrb,
data_in_cnvlia=>cnvlia, data_in_cnvlib=>cnvlib, clk=>clk_in,
clk_enable=>clk_enable_in, reset=>reset_in, dataI_out=>It_out,
dataQ_out=>Qt_out);

end structure;
----------------------------------------------------------------------
```

```
-- ---------------------------------------------------------------
-- Design: sss_sequ (varied)
-- To generate: tsui_sss_converter_struct_pipeline_dig
-- Original:
-- Re-Generated by Jesse P Somann
-- Last edited: 06 Feb 06
-- Purpose: I/Q data conversion
-- ---------------------------------------------------------------


-- ---------------------------------------------------------------
-- ---------------------------------------------------------------
-- Register1
-- ---------------------------------------------------------------
-- ---------------------------------------------------------------

----http://www.csee.umbc.edu/help/VHDL/samples/reg_g.vhdl

library ieee;
use ieee.std_logic_1164.all;
entity reg1 is
  port  (clk    : in  std_logic;
         input  : in  std_logic;
         output : out std_logic);
end entity reg1;

architecture behavior of reg1 is
begin  -- behavior
  reg: process(clk)
       begin
         if clk='1' then                -- rising edge
           output <= input;
         end if;
       end process reg;
end architecture behavior;

-- ---------------------------------------------------------------
-- ---------------------------------------------------------------
-- Register
-- ---------------------------------------------------------------
-- ---------------------------------------------------------------

----http://www.csee.umbc.edu/help/VHDL/samples/reg_g.vhdl

library ieee;
use ieee.std_logic_1164.all;
entity reg is
  generic(left   : natural);      -- top bit
  port  (clk    : in  std_logic;
         input  : in  std_logic_vector (left downto 0);
         output : out std_logic_vector (left downto 0) );
end entity reg;

architecture behavior of reg is
begin  -- behavior
  reg: process(clk)
       begin
         if clk='1' then                -- rising edge
           output <= input;
```

```
            end if;
        end process reg;
end architecture behavior;


-- ------------------------------------------------------------
-- ------------------------------------------------------------
-- Mux 2-way
-- ------------------------------------------------------------
-- ------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
entity sel2 is
  generic(left   : natural);      -- top bit
  port   (sel    : in  std_logic;
          input0 : in  std_logic_vector (left downto 0);
           input1 : in  std_logic_vector (left downto 0);
            output : out std_logic_vector (left downto 0) );
end entity sel2;

architecture behavior of sel2 is
begin  -- behavior
  sel_2: process(sel, input0, input1)
       begin
          if sel='0' then
            output <= input0;
         else
            output <= input1;
          end if;
        end process sel_2;
end architecture behavior;


-- ------------------------------------------------------------
-- ------------------------------------------------------------
-- Lookahead Adder for signed addition
-- ------------------------------------------------------------
-- ------------------------------------------------------------


-- ------------------------------------------------------------
-- Module: add_signed_lookahead_first (behavioral)
-- Original: http://www.altera.com/support/examples/vhdl/v_cl_addr.html
-- Re-Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: Signed carry lookahead adder, adjustable bit-widths
-- ------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY add_signed_lookahead_first IS
      GENERIC( bit_width_in                :   natural;
              bit_width_out                :   natural;
              bit_string                 :   std_logic_vector;
              bit_string_cii               :    std_logic_vector;
              pad_string                 :   std_logic_vector;
              pad_width                  :   natural);
      PORT (
              a:              IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
              b:              IN STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0);
              result:       OUT STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0));
END add_signed_lookahead_first;
```

```
ARCHITECTURE behavioral OF add_signed_lookahead_first IS

SIGNAL    a_pad             :    STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
         := bit_string;
SIGNAL    b_pad             :    STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
         := bit_string;
SIGNAL    h_sum             :    STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
         := bit_string;
SIGNAL    carry_generate    :    STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
         := bit_string;
SIGNAL    carry_propagate   :    STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 0)
         := bit_string;
SIGNAL    carry_in_internal :    STD_LOGIC_VECTOR((bit_width_out-1) DOWNTO 1)
         := bit_string_cii;

BEGIN

    PAD:PROCESS (a,b)
              variable a_int:            STD_LOGIC_VECTOR((bit_width_out - 1)
downto 0);
              variable b_int:            STD_LOGIC_VECTOR((bit_width_out - 1)
downto 0);
        BEGIN
              a_int := pad_string & (a);
              b_int := (b);
              for pad in 0 to (pad_width-1) loop
                    a_int((bit_width_in) + pad) := a_int(bit_width_in - 1);
              end loop;
              a_pad <= STD_LOGIC_VECTOR(a_int);
              b_pad <= STD_LOGIC_VECTOR(b_int);
    END PROCESS PAD;

    h_sum <= a_pad XOR b_pad;
    carry_generate <= a_pad AND b_pad;
    carry_propagate <= a_pad OR b_pad;

    PROCESS (carry_generate,carry_propagate,carry_in_internal)
            BEGIN
            carry_in_internal(1) <= carry_generate(0);
              inst: FOR i IN 1 TO (bit_width_out-2) LOOP
                    carry_in_internal(i+1) <= carry_generate(i) OR
(carry_propagate(i) AND carry_in_internal(i));
                  END LOOP;
--        result(bit_width_out-1) <= carry_generate(bit_width_in-1) OR
(carry_propagate(bit_width_in-1) AND carry_in_internal(bit_width_in-1));
    END PROCESS;

    result(0) <= h_sum(0);
    result((bit_width_out-1) DOWNTO 1) <= h_sum((bit_width_out-1) DOWNTO 1) XOR
carry_in_internal((bit_width_out-1) DOWNTO 1);
END behavioral;


-- ----------------------------------------------------------------
-- ----------------------------------------------------------------
-- Input data formation (coeff and input) for filter filtat
-- ----------------------------------------------------------------
-- ----------------------------------------------------------------


-- ----------------------------------------------------------------
-- Module: filtat_data
```

```vhdl
-- Original: Generated by MATLAB(R) 7.0.4 and the Filter Design HDL Coder 1.2.
-- Re-Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: Low pass filter for steaming data input.
-- ----------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;
--USE ieee.std_logic_signed.all;

ENTITY filtat_data IS

   GENERIC( max_flength                :   natural;
            bit_width_in        :   natural;
            bit_width_coeff     :   natural;
            bit_string_in       :   std_logic_vector;
            bit_string_coeff         :    std_logic_vector);

   PORT( clk                             :   IN    std_logic;
         clk_enable                      :   IN    std_logic;
         reset                           :   IN    std_logic;
         filter_in                       :   IN    std_logic_vector
(bit_width_in-1 downto 0);
         pipeline_data                     :    OUT   std_logic_vector
(bit_width_in-1 downto 0);
         coeff_data                  :    OUT   std_logic_vector (bit_width_coeff-
1 downto 0);
         ready                         :    OUT   std_logic);

END filtat_data;
-----------------------------------------------------------
--Module Architecture: filtat_data
-----------------------------------------------------------
ARCHITECTURE rtl OF filtat_data IS
  -- Local Functions

  -- Type Definitions
  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_in-1 downto 0);
  TYPE coeff_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_coeff-1 downto 0);

  -- Constants
  CONSTANT coeff                    : coeff_pipeline_type(1 TO max_flength) := (
   --cofat--

  -- Signals
  SIGNAL delay_pipeline                     : delay_pipeline_type(0 TO
max_flength-1);-- := (others => bit_string_in);
  SIGNAL pipeline_input                     : std_logic_vector (bit_width_in-1
downto 0);-- := bit_string_in;
  SIGNAL coeff_input                        : std_logic_vector (bit_width_coeff-1
downto 0);-- := bit_string_coeff;

BEGIN

  -- Block Statements
  Delay_Pipeline_process : PROCESS (clk, reset)
```

```vhdl
  variable cnt: integer range 0 to max_flength + 1;
  variable ready_check : std_logic;
  variable extra : std_logic;

  BEGIN

  IF reset = '1' THEN
              delay_pipeline(0 TO (max_flength-1)) <= (OTHERS => bit_string_in);
              pipeline_data <= bit_string_in;
              coeff_data <= bit_string_coeff;
              cnt := max_flength + 1;
              ready_check := '0';
              ready <= '0';
              extra := '0';

   ELSIF (rising_edge(clk)) then
       IF clk_enable = '1' THEN

              IF cnt > 1 THEN
                  pipeline_data <= delay_pipeline(max_flength - cnt + 1);
                  coeff_data <= coeff(max_flength - cnt + 2);
                  cnt := cnt - 1;
              END IF;

              IF cnt = 1 THEN
                  cnt := cnt - 1;
              END IF;

              IF cnt = 0 and ready_check = '1' THEN
                  pipeline_data <= bit_string_in;
                  coeff_data <= bit_string_coeff;
                  cnt := max_flength + 1;
                  ready_check := '0';
                  ready <= '0';
              END IF;

              IF cnt = 0 and ready_check = '0'then
                  ready_check := '1';
                  ready <= '1';
                  delay_pipeline(0) <= filter_in;
                  delay_pipeline(1 TO (max_flength-1)) <= delay_pipeline(0 TO
(max_flength-2));
              END IF;

       end if;
     end if;
  END PROCESS Delay_Pipeline_process;
END rtl;

-- ----------------------------------------------------------------
-- ----------------------------------------------------------------
-- Input data formation (coeff and input) for filter filtbt
-- ----------------------------------------------------------------
-- ----------------------------------------------------------------


-- ----------------------------------------------------------------
-- Module: filtbt_data
-- Original: Generated by MATLAB(R) 7.0.4 and the Filter Design HDL Coder 1.2.
-- Re-Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: Low pass filter for steaming data input.
```

189

```vhdl
-- ---------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;
--USE ieee.std_logic_signed.all;

ENTITY filtbt_data IS

   GENERIC( max_flength                  :   natural;
            bit_width_in        :    natural;
            bit_width_coeff     :    natural;
            bit_string_in       :    std_logic_vector;
            bit_string_coeff          :    std_logic_vector);

   PORT( clk                             :   IN    std_logic;
         clk_enable                      :   IN    std_logic;
         reset                           :   IN    std_logic;
         filter_in                       :   IN    std_logic_vector
(bit_width_in-1 downto 0);
         pipeline_data                        :   OUT   std_logic_vector
(bit_width_in-1 downto 0);
         coeff_data                   :   OUT   std_logic_vector (bit_width_coeff-
1 downto 0);
         ready                         :   OUT   std_logic);

END filtbt_data;
----------------------------------------------------------------
--Module Architecture: filtbt_data
----------------------------------------------------------------
ARCHITECTURE rtl OF filtbt_data IS
  -- Local Functions

  -- Type Definitions
  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_in-1 downto 0);
  TYPE coeff_pipeline_type IS ARRAY (NATURAL range <>) OF std_logic_vector
(bit_width_coeff-1 downto 0);

  -- Constants
  CONSTANT coeff                    : coeff_pipeline_type(1 TO max_flength) := (
   --cofbt--

  -- Signals
  SIGNAL delay_pipeline                     : delay_pipeline_type(0 TO
max_flength-1);-- := (others => bit_string_in);
  SIGNAL pipeline_input                 : std_logic_vector (bit_width_in-1
downto 0);-- := bit_string_in;
  SIGNAL coeff_input                    : std_logic_vector (bit_width_coeff-1
downto 0);-- := bit_string_coeff;

BEGIN

  -- Block Statements
  Delay_Pipeline_process : PROCESS (clk, reset)

  variable cnt: integer range 0 to max_flength + 1;
  variable ready_check : std_logic;
  variable extra : std_logic;
```

```
   BEGIN

   IF reset = '1' THEN
               delay_pipeline(0 TO (max_flength-1)) <= (OTHERS => bit_string_in);
               pipeline_data <= bit_string_in;
               coeff_data <= bit_string_coeff;
               cnt := max_flength + 1;
               ready_check := '0';
               ready <= '0';
               extra := '0';

    ELSIF (rising_edge(clk)) then
        IF clk_enable = '1' THEN

               IF cnt > 1 THEN
                   pipeline_data <= delay_pipeline(max_flength - cnt + 1);
                   coeff_data <= coeff(max_flength - cnt + 2);
                   cnt := cnt - 1;
               END IF;

               IF cnt = 1 THEN
                   cnt := cnt - 1;
               END IF;

               IF cnt = 0 and ready_check = '1' THEN
                   pipeline_data <= bit_string_in;
                   coeff_data <= bit_string_coeff;
                   cnt := max_flength + 1;
                   ready_check := '0';
                   ready <= '0';
               END IF;

               IF cnt = 0 and ready_check = '0'then
                   ready_check := '1';
                   ready <= '1';
                   delay_pipeline(0) <= filter_in;
                   delay_pipeline(1 TO (max_flength-1)) <= delay_pipeline(0 TO
(max_flength-2));
               END IF;

        end if;
    end if;
  END PROCESS Delay_Pipeline_process;
END rtl;




-- ----------------------------------------------------------------
-- ----------------------------------------------------------------
-- Booth multiplier in sequential form.
-- ----------------------------------------------------------------
-- ----------------------------------------------------------------


-- ----------------------------------------------------------------
-- Module: mult_booth (behavioral--sequential)
-- Original: Mark Zwolinski
--            : Department of Electronics and Computer Science
--            : University of Southampton
--            : Southampton SO17 1BJ, UK
--            : mz@ecs.soton.ac.uk
```

```vhdl
-- Re-Generated by Jesse P Somann
-- Last edited: 7 Feb 06
-- Purpose: Signed sequential multiplier
-- -------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

entity mult_booth is
        GENERIC( bit_width_in1                  :   natural;  --in1 <= in2
                 bit_width_in2                  :   natural;
                 bit_width_out                  :   natural;
                 bit_width_out_fin         :   natural;
                 bit_string_in                  :   std_logic_vector;
                 bit_string_out                 :   std_logic_vector;
                 pad_string                 :   std_logic_vector);
        PORT (
                clk:                IN  std_logic;
                clk_enable:  in  std_logic;
                reset:               in  std_logic;
                a:               IN STD_LOGIC_VECTOR ((bit_width_in1-1) DOWNTO 0);
                b:               IN STD_LOGIC_VECTOR ((bit_width_in2-1) DOWNTO 0);
                result:        OUT STD_LOGIC_VECTOR ((bit_width_out_fin-1) DOWNTO
0);
                ready:         OUT  std_logic);
end mult_booth;

architecture rtl of mult_booth is

SIGNAL    a_pad            :    STD_LOGIC_VECTOR((bit_width_in1-1) DOWNTO 0)
        := bit_string_in;
SIGNAL    b_pad            :    STD_LOGIC_VECTOR((bit_width_in1-1) DOWNTO 0)
        := bit_string_in;

begin

  PAD:PROCESS (a,b)
        variable a_int:                 STD_LOGIC_VECTOR((bit_width_in1 - 1)
downto 0);
        variable b_int:            STD_LOGIC_VECTOR((bit_width_in1 - 1) downto
0);
        BEGIN
          if (bit_width_in1 - bit_width_in2) > 0 then
                a_int := (a);
                b_int := pad_string & (b);
                for pad in 0 to ((bit_width_in1 - bit_width_in2) - 1) loop
                        b_int((bit_width_in2) + pad) := b_int(bit_width_in2 - 1);
                end loop;
          else
                a_int:= (a);
                b_int:= (b);
          end if;
          a_pad <= STD_LOGIC_VECTOR(a_int);
          b_pad <= STD_LOGIC_VECTOR(b_int);
  END PROCESS PAD;


  Main:process (clk)
    variable count : integer range 0 to bit_width_in1;
```

192

```vhdl
    variable pa : std_logic_vector((bit_width_in1 + bit_width_in1) downto 0);
    variable a_1 : std_logic;
    variable ready_check : std_logic;
    alias p : std_logic_vector(bit_width_in1 downto 0) is pa((bit_width_in1 +
bit_width_in1) downto bit_width_in1);
  begin
    if (rising_edge(clk)) then
       IF clk_enable = '1' THEN

         if reset = '1' then
           p := (others => '0');
           pa(bit_width_in1-1 downto 0) := (a_pad);
           a_1 := '0';
           count := bit_width_in1;
           ready_check := '0';
           ready <= '0';
         elsif count > 0 then
           case std_logic_vector'(pa(0), a_1) is
             when "01" =>
              p := p + (b_pad);
             when "10" =>
              p := p - (b_pad);
             when others => null;
           end case;
           a_1 := pa(0);
           pa((bit_width_in1 + bit_width_in1)-1 downto 0) := pa((bit_width_in1 +
bit_width_in1) downto 1);
           count := count - 1;
         end if;

       if count = 0 then
           ready <= '1';
           result <= std_logic_vector(pa(bit_width_in1 + bit_width_in2-1 downto
0));
         end if;

       end if;
    end if;
  end process Main;

end rtl;


-- --------------------------------------------------------------
-- --------------------------------------------------------------
-- Input Data Splitter for incoming data stream
-- --------------------------------------------------------------
-- --------------------------------------------------------------


-- --------------------------------------------------------------
-- Module: I/Qgen_dig
-- Generated by Jesse P Somann
-- Last edited: 30 Nov 06
-- Purpose: to split the input data into a real (I) and
--  imaginary (Q) stream initiative to be convolved
-- --------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
--use ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;
```

```vhdl
ENTITY I/Qgen_dig IS

    GENERIC( bit_width                       :    natural;
            bit_string                       :    std_logic_vector);

    PORT( clk                                :    IN    std_logic;
          clk_enable                         :    IN    std_logic;
          reset                              :    IN    std_logic;
          data_in                            :    IN    std_logic_vector (bit_width-
1  downto 0);
          dataI_out                          :    OUT   std_logic_vector (bit_width-
1  downto 0);
          dataQ_out                     :    OUT   std_logic_vector (bit_width-1
downto 0)
          );
END I/Qgen_dig;
-----------------------------------------------------------------
--Module Architecture: I/Qgen_dig
-----------------------------------------------------------------
ARCHITECTURE rtl OF I/Qgen_dig IS
 -- Local Functions

  -- Signals
  SIGNAL output_registerI                    : std_logic_vector (bit_width-1
downto 0) := bit_string;
  SIGNAL output_registerQ                    : std_logic_vector (bit_width-1
downto 0) := bit_string;

  SIGNAL counter4                     : std_logic_vector (1 downto 0);
  SIGNAL inverted                     : std_logic_vector (bit_width-1  downto 0)
:= bit_string;

BEGIN

  Output_Register_process : PROCESS (clk, reset)
  BEGIN

    IF reset = '1' THEN
      output_registerI <= bit_string;
      output_registerQ <= bit_string;
      counter4 <= "00";

    ELSIF clk'event AND clk = '1' THEN

      IF clk_enable = '1' THEN

        CASE counter4 is
              when "00" =>
                output_registerI <= data_in;
              when "01" =>
                output_registerQ <= data_in;
              when "10" =>
                output_registerI <= (not(data_in)) + '1';
              when "11" =>
                output_registerQ <= (not(data_in)) + '1';
              when others =>
                output_registerI <= bit_string;
                output_registerQ <= bit_string;
        END CASE;
        counter4 <= counter4 + "01";
```

194

```vhdl
      END IF;
    END IF;
  END PROCESS Output_Register_process;

  -- Assignment Statements
  dataI_out <= output_registerI;
  dataQ_out <= output_registerQ;

END rtl;



-- -------------------------------------------------------------
-- -------------------------------------------------------------
-- Filter 'a', low pass, sequential
-- -------------------------------------------------------------
-- -------------------------------------------------------------


-- -------------------------------------------------------------
-- Module: I/Qfir_filtat_dig_struct2_sequ
-- Original: Generated by MATLAB(R) 7.0.4 and the Filter Design HDL Coder 1.2.
-- Re-Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: Low pass filter for steaming data input.
-- -------------------------------------------------------------
-- This design is sequential and will take (bit_with_in(max)+1)*(max_flength+1)
clock cycles:
--    for example.... input = 6 bit and filter length of 16 gives 7*17= 119
cycles/operation
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
--USE IEEE.numeric_std.ALL;
--use ieee.std_logic_arith.all;
--USE ieee.std_logic_signed.all;

ENTITY I/Qfir_filtat_dig_struct_sequ IS

    GENERIC( max_flength               :   natural;
             levels              :   natural;
             bit_width_in        :   natural;
             bit_width_coeff     :   natural;
             bit_width_out       :   natural;
             bit_string_in       :   std_logic_vector;
             bit_string_coeff        :   std_logic_vector;
             bit_string_in_coeff     :   std_logic_vector;
             bit_string_out      :   std_logic_vector;
             bit_string_cii      :   std_logic_vector;
             pad_string              :   std_logic_vector;
             pad_string_mult     :   std_logic_vector;
             pad_width           :   natural);
    PORT( clk                         :   IN    std_logic;
          clk_enable                  :   IN    std_logic;
          reset                       :   IN    std_logic;
          filter_in                   :   IN    std_logic_vector
(bit_width_in-1 downto 0);
          filter_out                  :   OUT   std_logic_vector
(bit_width_out-1 downto 0);
          ready                       :   OUT   std_logic);
```

```
END I/Qfir_filtat_dig_struct_sequ;
-------------------------------------------------------------
--Module Architecture: I/Qfir_filtat_dig_struct_sequ
-------------------------------------------------------------
ARCHITECTURE rtl OF I/Qfir_filtat_dig_struct_sequ IS
  -- Local Functions

  -- Components
  component filtat_data
      GENERIC( max_flength                     :   natural;
               bit_width_in                    :   natural;
                bit_width_coeff        :   natural;
               bit_string_in                   :   std_logic_vector;
               bit_string_coeff        :   std_logic_vector);
      PORT( clk                                :   IN   std_logic;
            clk_enable                         :   IN   std_logic;
            reset                              :   IN   std_logic;
            filter_in                          :   IN   std_logic_vector
(bit_width_in-1 downto 0);
            pipeline_data                      :   OUT   std_logic_vector
(bit_width_in-1 downto 0);
            coeff_data                         :   OUT   std_logic_vector
(bit_width_coeff-1 downto 0);
            ready                        :   OUT   std_logic
            );
    end component;

    component add_signed_lookahead_first
      GENERIC( bit_width_in                    :   natural;
               bit_width_out                   :   natural;
               bit_string                 :   std_logic_vector;
               bit_string_cii             :    std_logic_vector;
               pad_string                 :   std_logic_vector;
               pad_width                  :   natural);
      PORT (
            a:              IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
            b:              IN STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0);
            result:         OUT STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0));
  end component;

  component mult_booth
      GENERIC( bit_width_in1                    :   natural;  --in1 <= in2
               bit_width_in2                    :   natural;
               bit_width_out                    :   natural;
               bit_width_out_fin        :   natural;
               bit_string_in                    :   std_logic_vector;
               bit_string_out                   :   std_logic_vector;
               pad_string                 :   std_logic_vector);
      PORT ( clk:              IN  std_logic;
             clk_enable:  in  std_logic;
             reset:              in  std_logic;
             a:              IN STD_LOGIC_VECTOR ((bit_width_in1-1) DOWNTO 0);
             b:              IN STD_LOGIC_VECTOR ((bit_width_in2-1) DOWNTO 0);
             result:         OUT STD_LOGIC_VECTOR ((bit_width_out_fin-1) DOWNTO
0);
             ready:          OUT  std_logic);
  end component;

  component sel2
      generic(left   : natural);        -- top bit
      port    (sel    : in  std_logic;
```

```vhdl
            input0  : in  std_logic_vector (left-1 downto 0);
            input1  : in  std_logic_vector (left-1 downto 0);
            output : out std_logic_vector (left-1 downto 0) );
  end component;

  component reg
      generic(left    : natural);      -- top bit
      port   (clk    : in  std_logic;
              input  : in  std_logic_vector (left downto 0);
              output : out std_logic_vector (left downto 0) );
  end component;

  component reg1
      port   (clk    : in  std_logic;
              input  : in  std_logic;
              output : out std_logic);
  end component;

  -- Signals
  SIGNAL pipeline                           : std_logic_vector (bit_width_in-1
downto 0);-- := bit_string_in;
  SIGNAL coeff                              : std_logic_vector (bit_width_coeff-1
downto 0);-- := bit_string_coeff;

  SIGNAL product                   : std_logic_vector (bit_width_in +
bit_width_coeff - 1 downto 0);-- := bit_string_in_coeff;
  SIGNAL sum                         : std_logic_vector (bit_width_out-1
downto 0);-- := bit_string_out;
  SIGNAL sum_mux                     : std_logic_vector (bit_width_out-1
downto 0);-- := bit_string_out;
  SIGNAL sum_reg                      : std_logic_vector (bit_width_out-1 downto
0);-- := bit_string_out;

  SIGNAL mux_sel                     : std_logic;
  SIGNAL mux_sel_reg                       : std_logic;

  SIGNAL mult_ready                  : std_logic;
  SIGNAL mult_reset                  : std_logic;

  SIGNAL clk_long                    : std_logic;

  SIGNAL output_register                   : std_logic_vector (bit_width_out-1
downto 0);-- := bit_string_out;


BEGIN

  clk_ext:process (clk)
    variable count : integer range 0 to bit_width_in + 1;
  begin
    if (rising_edge(clk)) then
      IF clk_enable = '1' THEN
        if reset = '1' then
          count := bit_width_in + 1;
          clk_long <= '1';
        elsif count > 0 then
          clk_long <= '0';
          count := count - 1;
        end if;

        if count = 0 then
```

```vhdl
          count := bit_width_in + 1;
          clk_long <= '1';
        end if;
      end if;
    end if;
  end process clk_ext;
  ----------------------------------------------------------------------
  ------

   clocked_data: filtat_data generic map(max_flength=>max_flength,
bit_width_in=>bit_width_in, bit_width_coeff=>bit_width_coeff,
bit_string_in=>bit_string_in, bit_string_coeff=>bit_string_coeff)
                          port map(clk=>clk_long, clk_enable=>clk_enable,
reset=>reset, filter_in=>filter_in, pipeline_data=>pipeline, coeff_data=>coeff,
ready=>mux_sel);


   mult_op_1: IF (bit_width_in >= bit_width_coeff) generate
             multiplier1: mult_booth generic map (bit_width_in1=>bit_width_in,
bit_width_in2=>bit_width_coeff, bit_width_out=>(bit_width_in + bit_width_in),
bit_width_out_fin=>(bit_width_in + bit_width_coeff),
bit_string_in=>bit_string_in, bit_string_out=>bit_string_in_coeff,
pad_string=>pad_string_mult)
                        port map (clk=>clk, clk_enable=>clk_enable,
reset=>mult_reset, a=>pipeline, b=>coeff, result=>product, ready=>mult_ready);
        END generate mult_op_1;
   mult_op_2: IF (bit_width_in < bit_width_coeff) generate
             multiplier2: mult_booth generic map
(bit_width_in1=>bit_width_coeff, bit_width_in2=>bit_width_in,
bit_width_out=>(bit_width_in + bit_width_in), bit_width_out_fin=>(bit_width_in
+ bit_width_coeff), bit_string_in=>bit_string_coeff,
bit_string_out=>bit_string_in_coeff, pad_string=>pad_string_mult)
                        port map (clk=>clk, clk_enable=>clk_enable,
reset=>mult_reset, a=>coeff, b=>pipeline, result=>product, ready=>mult_ready);
        END generate mult_op_2;

   add_reg: reg generic map (left=>(bit_width_out -1))
             port map(clk=>clk_long, input=>sum, output=>sum_reg);

   ready_reg: reg1 port map(clk=>clk_long, input=>mux_sel,
output=>mux_sel_reg);

   add_in_select: sel2 generic map (left=>(bit_width_out - 1))
                     port map(sel=>mux_sel_reg, input0=>sum_reg,
input1=>bit_string_out, output=>sum_mux);

   adder: add_signed_lookahead_first generic map (bit_width_in=>(bit_width_in +
bit_width_coeff), bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii, pad_string=>pad_string, pad_width=>pad_width)
                               port map(a=>product, b=>sum_mux, result=>sum);

  ----------------------------------------------------------------------
  ------
  mult_rdy: PROCESS (reset, mult_ready)
  BEGIN
    if reset = '1' or mult_ready = '1' then
     mult_reset <= '1';
    else
     mult_reset <= '0';
    end if;
  END PROCESS mult_rdy;
```

```vhdl
  filt_out: PROCESS (mux_sel_reg)
  BEGIN
    if mux_sel_reg = '1' then
     filter_out <= sum;
    end if;
  END PROCESS filt_out;

END rtl;


-- ------------------------------------------------------------
-- ------------------------------------------------------------
-- Filter 'b', low pass, sequential
-- ------------------------------------------------------------
-- ------------------------------------------------------------


-- ------------------------------------------------------------
-- Module: I/Qfir_filtbt_dig_struct2_sequ
-- Original: Generated by MATLAB(R) 7.0.4 and the Filter Design HDL Coder 1.2.
-- Re-Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: Low pass filter for steaming data input.
-- ------------------------------------------------------------
-- This design is sequential and will take (bit_with_in(max)+1)*(max_flength+1)
clock cycles:
--    for example.... input = 6 bit and filter length of 16 gives 7*17= 119
cycles/operation
-- ------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
--USE IEEE.numeric_std.ALL;
--use ieee.std_logic_arith.all;
--USE ieee.std_logic_signed.all;

ENTITY I/Qfir_filtbt_dig_struct_sequ IS

   GENERIC( max_flength                 :   natural;
           levels               :   natural;
           bit_width_in         :   natural;
           bit_width_coeff      :   natural;
           bit_width_out        :   natural;
           bit_string_in        :   std_logic_vector;
           bit_string_coeff         :   std_logic_vector;
           bit_string_in_coeff      :   std_logic_vector;
           bit_string_out       :   std_logic_vector;
           bit_string_cii       :   std_logic_vector;
           pad_string               :   std_logic_vector;
           pad_string_mult      :   std_logic_vector;
           pad_width            :   natural);
   PORT( clk                        :   IN    std_logic;
        clk_enable              :   IN    std_logic;
        reset                   :   IN    std_logic;
        filter_in               :   IN    std_logic_vector
(bit_width_in-1 downto 0);
        filter_out                  :   OUT   std_logic_vector
(bit_width_out-1 downto 0);
        ready                       :   OUT   std_logic);

END I/Qfir_filtbt_dig_struct_sequ;
```

199

```vhdl
--------------------------------------------------------------
--Module Architecture: I/Qfir_filtbt_dig_struct_sequ
--------------------------------------------------------------
ARCHITECTURE rtl OF I/Qfir_filtbt_dig_struct_sequ IS
  -- Local Functions

  -- Components
  component filtbt_data
      GENERIC( max_flength                    :   natural;
               bit_width_in                   :   natural;
               bit_width_coeff          :   natural;
               bit_string_in                  :   std_logic_vector;
               bit_string_coeff         :   std_logic_vector);
      PORT( clk                               :   IN    std_logic;
            clk_enable                        :   IN    std_logic;
            reset                             :   IN    std_logic;
            filter_in                         :   IN    std_logic_vector
(bit_width_in-1 downto 0);
            pipeline_data                     :   OUT   std_logic_vector
(bit_width_in-1 downto 0);
            coeff_data                        :   OUT   std_logic_vector
(bit_width_coeff-1 downto 0);
            ready                      :   OUT   std_logic
            );
    end component;

    component add_signed_lookahead_first
      GENERIC( bit_width_in                   :   natural;
               bit_width_out                  :   natural;
               bit_string               :   std_logic_vector;
               bit_string_cii                 :   std_logic_vector;
               pad_string               :   std_logic_vector;
               pad_width                :   natural);
      PORT (
            a:             IN STD_LOGIC_VECTOR ((bit_width_in-1) DOWNTO 0);
            b:             IN STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0);
            result:        OUT STD_LOGIC_VECTOR ((bit_width_out-1) DOWNTO 0));
    end component;

  component mult_booth
      GENERIC( bit_width_in1                  :   natural;  --in1 <= in2
               bit_width_in2                  :   natural;
               bit_width_out                  :   natural;
               bit_width_out_fin        :   natural;
               bit_string_in                  :   std_logic_vector;
               bit_string_out                 :   std_logic_vector;
               pad_string               :   std_logic_vector);
      PORT ( clk:              IN  std_logic;
            clk_enable:  in  std_logic;
            reset:            in  std_logic;
            a:             IN STD_LOGIC_VECTOR ((bit_width_in1-1) DOWNTO 0);
            b:             IN STD_LOGIC_VECTOR ((bit_width_in2-1) DOWNTO 0);
            result:        OUT STD_LOGIC_VECTOR ((bit_width_out_fin-1) DOWNTO
0);
            ready:         OUT  std_logic);
    end component;

  component sel2
      generic(left   : natural);       -- top bit
      port   (sel    : in  std_logic;
             input0 : in  std_logic_vector (left-1 downto 0);
```

200

```vhdl
            input1  : in  std_logic_vector (left-1 downto 0);
            output : out std_logic_vector (left-1 downto 0) );
    end component;

  component reg
      generic(left   : natural);     -- top bit
      port   (clk    : in  std_logic;
              input  : in  std_logic_vector (left downto 0);
              output : out std_logic_vector (left downto 0) );
    end component;

  component reg1
      port   (clk    : in  std_logic;
              input  : in  std_logic;
              output : out std_logic);
    end component;

  -- Signals
  SIGNAL pipeline                         : std_logic_vector (bit_width_in-1
downto 0);-- := bit_string_in;
  SIGNAL coeff                            : std_logic_vector (bit_width_coeff-1
downto 0);-- := bit_string_coeff;

  SIGNAL product                  : std_logic_vector (bit_width_in +
bit_width_coeff - 1 downto 0);-- := bit_string_in_coeff;
  SIGNAL sum                       : std_logic_vector (bit_width_out-1
downto 0);-- := bit_string_out;
  SIGNAL sum_mux                    : std_logic_vector (bit_width_out-1
downto 0);-- := bit_string_out;
  SIGNAL sum_reg                     : std_logic_vector (bit_width_out-1 downto
0);-- := bit_string_out;

  SIGNAL mux_sel                   : std_logic;
  SIGNAL mux_sel_reg                    : std_logic;

  SIGNAL mult_ready                : std_logic;
  SIGNAL mult_reset                : std_logic;

  SIGNAL clk_long                  : std_logic;

  SIGNAL output_register               : std_logic_vector (bit_width_out-1
downto 0);-- := bit_string_out;


BEGIN

  clk_ext:process (clk)
    variable count : integer range 0 to bit_width_in + 1;
  begin
    if (rising_edge(clk)) then
      IF clk_enable = '1' THEN
        if reset = '1' then
          count := bit_width_in + 1;
          clk_long <= '1';
        elsif count > 0 then
          clk_long <= '0';
          count := count - 1;
        end if;

        if count = 0 then
          count := bit_width_in + 1;
```

```vhdl
         clk_long <= '1';
        end if;
      end if;
    end if;
  end process clk_ext;
  ----------------------------------------------------------------------------
------

   clocked_data: filtbt_data generic map(max_flength=>max_flength,
bit_width_in=>bit_width_in, bit_width_coeff=>bit_width_coeff,
bit_string_in=>bit_string_in, bit_string_coeff=>bit_string_coeff)
                        port map(clk=>clk_long, clk_enable=>clk_enable,
reset=>reset, filter_in=>filter_in, pipeline_data=>pipeline, coeff_data=>coeff,
ready=>mux_sel);


   mult_op_1: IF (bit_width_in >= bit_width_coeff) generate
            multiplier1: mult_booth generic map (bit_width_in1=>bit_width_in,
bit_width_in2=>bit_width_coeff, bit_width_out=>(bit_width_in + bit_width_in),
bit_width_out_fin=>(bit_width_in + bit_width_coeff),
bit_string_in=>bit_string_in, bit_string_out=>bit_string_in_coeff,
pad_string=>pad_string_mult)
                     port map (clk=>clk, clk_enable=>clk_enable,
reset=>mult_reset, a=>pipeline, b=>coeff, result=>product, ready=>mult_ready);
       END generate mult_op_1;
   mult_op_2: IF (bit_width_in < bit_width_coeff) generate
            multiplier2: mult_booth generic map
(bit_width_in1=>bit_width_coeff, bit_width_in2=>bit_width_in,
bit_width_out=>(bit_width_in + bit_width_in), bit_width_out_fin=>(bit_width_in
+ bit_width_coeff), bit_string_in=>bit_string_coeff,
bit_string_out=>bit_string_in_coeff, pad_string=>pad_string_mult)
                     port map (clk=>clk, clk_enable=>clk_enable,
reset=>mult_reset, a=>coeff, b=>pipeline, result=>product, ready=>mult_ready);
       END generate mult_op_2;

   add_reg: reg generic map (left=>(bit_width_out -1))
             port map(clk=>clk_long, input=>sum, output=>sum_reg);

   ready_reg: reg1 port map(clk=>clk_long, input=>mux_sel,
output=>mux_sel_reg);

   add_in_select: sel2 generic map (left=>(bit_width_out - 1))
                   port map(sel=>mux_sel_reg, input0=>sum_reg,
input1=>bit_string_out, output=>sum_mux);

   adder: add_signed_lookahead_first generic map (bit_width_in=>(bit_width_in +
bit_width_coeff), bit_width_out=>bit_width_out, bit_string=>bit_string_out,
bit_string_cii=>bit_string_cii, pad_string=>pad_string, pad_width=>pad_width)
                        port map(a=>product, b=>sum_mux, result=>sum);

  ----------------------------------------------------------------------------
------
  mult_rdy: PROCESS (reset, mult_ready)
  BEGIN
    if reset = '1' or mult_ready = '1' then
     mult_reset <= '1';
    else
     mult_reset <= '0';
    end if;
  END PROCESS mult_rdy;
```

```
  filt_out: PROCESS (mux_sel_reg)
  BEGIN
    if mux_sel_reg = '1' then
     filter_out <= sum;
    end if;
  END PROCESS filt_out;

END rtl;


-- ------------------------------------------------------------
-- ------------------------------------------------------------
-- Recombination Unit for making final I and Q channels
-- ------------------------------------------------------------
-- ------------------------------------------------------------


-- ------------------------------------------------------------
-- Module: I/Qoutput_dig
-- Generated by Jesse P Somann
-- Last edited: 30 Nov 06
-- Purpose: To reform the output from 4 filters into I and Q data
--   streams
-- ------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
--USE IEEE.numeric_std.ALL;
--use ieee.std_logic_arith.all;
--USE ieee.std_logic_signed.all;
--use ieee.std_logic_unsigned.all;

ENTITY I/Qoutput_dig IS

   GENERIC( bit_width                      :   natural;
            bit_string                     :   std_logic_vector);

   PORT( clk                               :   IN    std_logic;
         clk_enable                        :   IN    std_logic;
         reset                             :   IN    std_logic;
         data_in_cnvlra                    :   IN    std_logic_vector (bit_width-
1 downto 0);
         data_in_cnvlrb                    :   IN    std_logic_vector (bit_width-
1 downto 0);
         data_in_cnvlia                    :   IN    std_logic_vector (bit_width-
1 downto 0);
         data_in_cnvlib                    :   IN    std_logic_vector (bit_width-
1 downto 0);
         dataI_out                         :   OUT   std_logic_vector (bit_width-
1 downto 0);
         dataQ_out                  :   OUT   std_logic_vector (bit_width-1
downto 0)
         );

END I/Qoutput_dig;
-- ------------------------------------------------------------
--Module Architecture: I/Qoutput_dig
-- ------------------------------------------------------------
ARCHITECTURE rtl OF I/Qoutput_dig IS
  -- Local Functions

  SIGNAL cnvli_delayed                     : std_logic_vector (bit_width-1
downto 0) := bit_string;
```

```vhdl
  SIGNAL output_registerI                      : std_logic_vector (bit_width-1
downto 0) := bit_string;
  SIGNAL output_registerQ                      : std_logic_vector (bit_width-1
downto 0) := bit_string;

  SIGNAL counter2                  : std_logic;
  SIGNAL q_zero                                : std_logic;
  SIGNAL clk2                      : std_logic;

BEGIN

  Output_Register_process : PROCESS (clk, reset)
  BEGIN

    IF reset = '1' THEN
      output_registerI <= bit_string;
      output_registerQ <= bit_string;
      q_zero <= '0';
      counter2 <= '0';

    ELSIF clk'event AND clk = '1' THEN

      IF clk_enable = '1' AND q_zero = '0' THEN
       output_registerI <= data_in_cnvlrb;
       output_registerQ <= bit_string;
       cnvli_delayed <= data_in_cnvlib;
       counter2 <= '1';
       q_zero <= '1';

      ELSIF clk_enable = '1' THEN
        CASE counter2 is
              when '0' =>
                output_registerI <= data_in_cnvlrb;
                output_registerQ <= cnvli_delayed;
                cnvli_delayed <= data_in_cnvlib;
              when '1' =>
                output_registerI <= data_in_cnvlra;
                output_registerQ <= cnvli_delayed;
                cnvli_delayed <= data_in_cnvlia;
              when others =>
                output_registerI <= bit_string;
                output_registerQ <= bit_string;
        END CASE;

        IF counter2 = '0' THEN
          counter2 <= '1'; END IF;
        IF counter2 = '1' THEN
          counter2 <= '0'; END IF;

      END IF;
    END IF;
  END PROCESS Output_Register_process;

  -- Assignment Statements
  dataI_out <= output_registerI;
  dataQ_out <= output_registerQ;

END rtl;
```

```
-- ------------------------------------------------------------
-- ------------------------------------------------------------
-- Toplevel Structural Setup
-- ------------------------------------------------------------
-- ------------------------------------------------------------


-- ------------------------------------------------------------
-- Module: tsui_sss_converter_struct2_sequ_dig
-- Generated by Jesse P Somann
-- Last edited: 30 Jan 06
-- Purpose: To convert input data stream into I/Q data streams
-- ------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity tsui_sss_converter_struct2_sequ_dig is
     GENERIC( max_flength        :   natural := --mfl--;
          levels                 :   natural := --lvs--;
          bit_width_in           :   natural := --bwi--;
          bit_width_coeff        :   natural := --bwc--;
          bit_width_mult_pad          :    natural := --bwmp--;
          bit_width_out          :   natural := --bwo--;
          bit_string_in          :   std_logic_vector (--bsin-- downto 0) := -
-bsi--; --"000000"
          bit_string_coeff              :   std_logic_vector (--bscn-- downto
0) := --bsc--;
          bit_string_in_coeff           :   std_logic_vector (--bsicn-- downto
0) := --bsic--;
          bit_string_out         :   std_logic_vector (--bson-- downto 0) := -
-bso--;
          bit_string_cii         :   std_logic_vector (--ciin-- downto 1) := -
-cii--;
          --psm--     --pad_string_mult
          pad_string_add         :   std_logic_vector (--psan-- downto 0) := -
-psa--;
          extension              :   natural := --ext--);
   PORT( clk_in                           :   IN    std_logic;
        clk_enable_in                     :   IN    std_logic;
        reset_in                          :   IN    std_logic;
        x_in                     :   IN    std_logic_vector (bit_width_in-
1 downto 0);
        It_out                            :    OUT    std_logic_vector
(bit_width_out-1 downto 0);
        Qt_out                            :    OUT    std_logic_vector
(bit_width_out-1 downto 0)
        );
end tsui_sss_converter_struct2_sequ_dig;


----------------------------------------------------------------
--Module Architecture: tsui_sss_converter_struct2_sequ_dig
----------------------------------------------------------------
architecture structure of tsui_sss_converter_struct2_sequ_dig is

component I/Qgen_dig
   GENERIC( bit_width                      :    natural;
          bit_string                      :    std_logic_vector);
   PORT( clk                              :    IN    std_logic;
        clk_enable                        :    IN    std_logic;
        reset                             :    IN    std_logic;
        data_in                           :    IN    std_logic_vector
(bit_width_in-1 downto 0);
```

```
        dataI_out                          :   OUT    std_logic_vector
(bit_width_in-1 downto 0);
        dataQ_out                  :   OUT   std_logic_vector (bit_width_in-1
downto 0)
        );
end component;

component I/Qfir_filtat_dig_struct_sequ
   GENERIC( max_flength                  :   natural;
         levels                 :   natural;
         bit_width_in           :   natural;
         bit_width_coeff        :   natural;
         bit_width_out          :   natural;
         bit_string_in          :   std_logic_vector;
         bit_string_coeff            :   std_logic_vector;
         bit_string_in_coeff         :   std_logic_vector;
         bit_string_out         :   std_logic_vector;
         bit_string_cii         :   std_logic_vector;
         pad_string                  :   std_logic_vector;
         pad_string_mult        :   std_logic_vector;
         pad_width              :   natural);
   PORT( clk                            :   IN    std_logic;
        clk_enable               :   IN    std_logic;
        reset                    :   IN    std_logic;
        filter_in                :   IN    std_logic_vector
(bit_width_in-1 downto 0);
        filter_out               :   OUT   std_logic_vector
(bit_width_out-1 downto 0);
        ready                    :   OUT   std_logic);
end component;

component I/Qfir_filtbt_dig_struct_sequ
   GENERIC( max_flength                  :   natural;
         levels                 :   natural;
         bit_width_in           :   natural;
         bit_width_coeff        :   natural;
         bit_width_out          :   natural;
         bit_string_in          :   std_logic_vector;
         bit_string_coeff            :   std_logic_vector;
         bit_string_in_coeff         :   std_logic_vector;
         bit_string_out         :   std_logic_vector;
         bit_string_cii         :   std_logic_vector;
         pad_string                  :   std_logic_vector;
         pad_string_mult        :   std_logic_vector;
         pad_width              :   natural);
   PORT( clk                            :   IN    std_logic;
        clk_enable               :   IN    std_logic;
        reset                    :   IN    std_logic;
        filter_in                :   IN    std_logic_vector
(bit_width_in-1 downto 0);
        filter_out               :   OUT   std_logic_vector
(bit_width_out-1 downto 0);
        ready                    :   OUT   std_logic);
end component;

component I/Qoutput_dig
   GENERIC( bit_width                    :   natural;
         bit_string                 :   std_logic_vector);
   PORT( clk                            :   IN    std_logic;
        clk_enable               :   IN    std_logic;
        reset                    :   IN    std_logic;
```

206

```vhdl
        data_in_cnvlra                    :   IN    std_logic_vector
(bit_width_out-1 downto 0);
        data_in_cnvlrb                    :   IN    std_logic_vector
(bit_width_out-1 downto 0);
        data_in_cnvlia                    :   IN    std_logic_vector
(bit_width_out-1 downto 0);
        data_in_cnvlib                    :   IN    std_logic_vector
(bit_width_out-1 downto 0);
        dataI_out                         :   OUT   std_logic_vector
(bit_width_out-1 downto 0);
        dataQ_out                 :   OUT   std_logic_vector (bit_width_out-1
downto 0)
        );
end component;

signal xr, xi:                             std_logic_vector (bit_width_in-1
downto 0);
signal cnvlra, cnvlrb, cnvlia, cnvlib:     std_logic_vector (bit_width_out-1
downto 0);
SIGNAL clk_long     :                           std_logic := '0';
SIGNAL clk2   :                         std_logic := '0';

begin

  clk2_process : PROCESS (clk_in)
  variable switch:              STD_LOGIC := '1';
  BEGIN
   IF clk_in'event AND clk_in = '1' AND switch = '1' THEN
      IF clk_enable_in = '1' THEN
        clk2 <= NOT clk2;
        switch := '0';
      END IF;
    END IF;
    IF clk_in'event AND clk_in = '1' AND switch = '0' THEN
      IF clk_enable_in = '1' THEN
        switch := '1';
      END IF;
    END IF;
  END PROCESS clk2_process;

  clk_ext:process (clk_in)
    variable count : integer range 0 to (extension + 1);
  begin
    if (rising_edge(clk_in)) then
      IF clk_enable_in = '1' THEN
        if reset_in = '1' then
          count := extension + 1;
          clk_long <= '1';
        elsif count > 0 then
          clk_long <= '0';
          count := count - 1;
        end if;

        if count = 0 then
          count := extension + 1;
          clk_long <= '1';
        end if;
      end if;
    end if;
  end process clk_ext;
```

```
     g1: I/Qgen_dig generic map (bit_width=>bit_width_in,
bit_string=>bit_string_in)
                    port map (data_in=>x_in, clk=>clk_long,
clk_enable=>clk_enable_in, reset=>reset_in, dataI_out=>xi, dataQ_out=>xr);
     g2: I/Qfir_filtat_dig_struct_sequ generic map (max_flength=>max_flength,
levels=>levels, bit_width_in=>bit_width_in, bit_width_coeff=>bit_width_coeff,
bit_width_out=>bit_width_out, bit_string_in=>bit_string_in,
bit_string_coeff=>bit_string_coeff, bit_string_in_coeff=>bit_string_in_coeff,
bit_string_out=>bit_string_out, bit_string_cii=>bit_string_cii,
pad_string=>pad_string_add, pad_string_mult=>pad_string_mult,
pad_width=>levels)
                                      port map (filter_in=>xr, clk=>clk2,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlra);
     g3: I/Qfir_filtat_dig_struct_sequ generic map (max_flength=>max_flength,
levels=>levels, bit_width_in=>bit_width_in, bit_width_coeff=>bit_width_coeff,
bit_width_out=>bit_width_out, bit_string_in=>bit_string_in,
bit_string_coeff=>bit_string_coeff, bit_string_in_coeff=>bit_string_in_coeff,
bit_string_out=>bit_string_out, bit_string_cii=>bit_string_cii,
pad_string=>pad_string_add, pad_string_mult=>pad_string_mult,
pad_width=>levels)
                                      port map (filter_in=>xi, clk=>clk2,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlia);
     g4: I/Qfir_filtbt_dig_struct_sequ generic map (max_flength=>max_flength,
levels=>levels, bit_width_in=>bit_width_in, bit_width_coeff=>bit_width_coeff,
bit_width_out=>bit_width_out, bit_string_in=>bit_string_in,
bit_string_coeff=>bit_string_coeff, bit_string_in_coeff=>bit_string_in_coeff,
bit_string_out=>bit_string_out, bit_string_cii=>bit_string_cii,
pad_string=>pad_string_add, pad_string_mult=>pad_string_mult,
pad_width=>levels)
                                      port map (filter_in=>xr, clk=>clk2,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlrb);
     g5: I/Qfir_filtbt_dig_struct_sequ generic map (max_flength=>max_flength,
levels=>levels, bit_width_in=>bit_width_in, bit_width_coeff=>bit_width_coeff,
bit_width_out=>bit_width_out, bit_string_in=>bit_string_in,
bit_string_coeff=>bit_string_coeff, bit_string_in_coeff=>bit_string_in_coeff,
bit_string_out=>bit_string_out, bit_string_cii=>bit_string_cii,
pad_string=>pad_string_add, pad_string_mult=>pad_string_mult,
pad_width=>levels)
                                      port map (filter_in=>xi, clk=>clk2,
clk_enable=>clk_enable_in, reset=>reset_in, filter_out=>cnvlib);
     g6: I/Qoutput_dig generic map (bit_width=>bit_width_out,
bit_string=>bit_string_out)
                    port map (data_in_cnvlra=>cnvlra, data_in_cnvlrb=>cnvlrb,
data_in_cnvlia=>cnvlia, data_in_cnvlib=>cnvlib, clk=>clk_long,
clk_enable=>clk_enable_in, reset=>reset_in, dataI_out=>Qt_out,
dataQ_out=>It_out);

end structure;
-------------------------------------------------------------------
```

Appendix P: coeff_gen.m (Matlab)

```matlab
% program to output formatted .txt files for Perl script use
clear;
format long;

%---------- FILTER PARAMETERS & GENERATION--(Kaiser) -----------------
dbst = 70;

if (dbst > 50)
    betat = .1102*(dbst-8.7);  % if dbst is > 50
elseif ((50 >= dbst) & (dbst >= 21))
    betat = .5842*(dbst-21)^0.4+.07886*(dbst-21);  % if 50 >= dbst >= 21
else
    betat = 0;  % if dbst is < 21
end;

% filter_length = 32;
filter_length = input('enter filter_length (even integer) = ');

windowt = kaiser(filter_length,betat);
% figure(1);
% wvtool(windowt)

total_bits = input('enter total coefficient bit_width = ');
fraction_bits = input('enter number of fractional coeff. bits = ');
% total_bits = 6;
% fraction_bits = 5;

hkt = fir1(filter_length-1, 0.5, windowt);

fi_hkt = fi(hkt, 1, total_bits, fraction_bits);

filtab = reshape(fi_hkt, 2, filter_length/2);
filtat = filtab(1,1:(filter_length/2));
filtbt = filtab(2,1:(filter_length/2));

%print out results into formatted text file
fid_coeff = fopen('filter_coeff.txt', 'wt');

fprintf(fid_coeff, '%i\n\n', filter_length/2);
fprintf(fid_coeff, '%i\n\n', total_bits);

for cnt = 1:1:((filter_length/2)-1)
    fprintf(fid_coeff, '"%s",\n',bin(filtat(cnt)));
end
fprintf(fid_coeff, '"%s");\n',bin(filtat(filter_length/2)));

fprintf(fid_coeff, '\n');

for cnt = 1:1:((filter_length/2)-1)
    fprintf(fid_coeff, '"%s",\n',bin(filtbt(cnt)));
end
fprintf(fid_coeff, '"%s");',bin(filtbt(filter_length/2)));

fclose(fid_coeff);
```

```matlab
%---------- INPUT DATA SIGNAL--FOR TESTBENCHES -----------------

n1 = 256;        % number of samples to be taken
n = [0:n1-1];
f = input('enter input frequency in MHz = ');
sf = input('enter sampling frequency in MHz, must be at least twice f = ');
%f = 14;
%sf = 100;
%form sine wave
x = sin(2*pi*(f*10^6)*((1/sf)*1e-6)*n+.01);

total_bits_in = input('enter total input bit_width = ');
fraction_bits_in = input('enter number of fractional input bits = ');
% total_bits_in = 6;
% fraction_bits_in = 4;

fi_x = fi(x,1,total_bits_in,fraction_bits_in);

fid_input = fopen('tb_input.txt', 'wt');
for cnt = 1:1:n1-1
    fprintf(fid_input, '"%s",\n',bin(fi_x(cnt)));
end
fprintf(fid_input, '"%s");\n',bin(fi_x(n1)));
fclose(fid_input);
```

Appendix Q: sss_gen.prl (Perl)

```perl
#!/usr/sbin/perl -w

#use 5.004;
#use strict;            # enforce declarations and quoting
#use CGI qw(:standard); # import shortcuts
#use Fcntl qw(:flock);  # imports LOCK_EX, LOCK_SH, LOCK_NB

my(
    $MATLAB_IN, # name of filter coefficient .m input file
    $STAT_NAME, # name input by user for stat file
    $STAT_OUT, # name of out puts stats file
    $PIPE_IN, # name of pipelined vhdl input file
    $PIPE_OUT, # name of pipelined vhdl output file
    $SEQU_IN, # name of sequential vhdl input file
    $SEQU_OUT, # name of sequential vhdl output file
    $ABST_IN, # name of abstract vhdl input file
    $ABST_OUT, # name of abstract vhdl output file

    @stats, # array to keep track of stats in

    @filt_info, # array to hold info of input Matlab txt file
    @cofat, #coefficient list for filtat for input to templates
    @cofbt, #coefficient list for filtbt for input to templates
    @pipe_in, #array holding pipeline template
    $pipe_in_l, #length of the array
    @sequ_in, #array holding sequential template
    $sequ_in_l, #length of the array
    @abst_in, #array holding abstract template
    $abst_in_l, #length of the array
    $filt_leng, # filter length (half of total filter length)
    $bw_in, # bit width of the input
    $bw_coeff, # bit width of the coefficients
    $bw_out, # bit width of the output

    # variables for input to template files
    $mfl, $lvs, $bwi, $bwc, $bwmp, $bwo, $bsi, $bsc, $bsic, $bso, $cii, $psm,
$psa, $ext,

    $i, # a counter variable for for loops
    $fill, # a fill element for reading coeff input file
    $first,

);

# usage: gen_sss.prl [matlab_file] [output_file] [-i] [bit_width_in] [-o]
[bit_with_out] [-s] [-p] [-a]
# $TITLE = "Replace template VHDL files with parameterized versions";
# This program will replace generic varibles in the templates with user defined
values for filter length and bit_widths.
# operation: "gen_sss.prl

# print ("\nEnter the name of the Original Fault File : ");


# Put input templates into info arrays
# open the input files for read (preserving old contents)
$PIPE_IN = "sss_pipeline_template.vhd";
chomp($PIPE_IN); #get rid of pesky \n
```

```perl
open(IN_PIPE, "< $PIPE_IN") || die("cannot open $PIPE_IN: $!");
@pipe_in=<IN_PIPE>;
close(IN_PIPE);

$SEQU_IN = "sss_sequential_template.vhd";
chomp($SEQU_IN); #get rid of pesky \n
open(IN_SEQU, "< $SEQU_IN") || die("cannot open $SEQU_IN: $!");
@sequ_in=<IN_SEQU>;
close(IN_SEQU);

#$ABST_IN = "sss_abstract_template.vhd";
#chomp($ABST_IN); #get rid of pesky \n
#open(IN_ABST, "< $ABST_IN") || die("cannot open $ABST_IN: $!");
#@abst_in=<IN_ABST>;
#close(IN_ABST);


# Read in first agument as input file and put into info array
$MATLAB_IN = $ARGV[0];
chomp($MATLAB_IN); #get rid of pesky \n
open(IN_MATLAB, "< $MATLAB_IN") || die("cannot open $MATLAB_IN: $!");
$filt_leng=<IN_MATLAB>;
$fill=<IN_MATLAB>;
$bw_coeff=<IN_MATLAB>;
$fill=<IN_MATLAB>;
@filt_info=<IN_MATLAB>;
close(IN_MATLAB);

# Split the filt_info array into two separate coefficient arrays
for($i=0;$i<$filt_leng;$i++){
    $cofat[$i] = $filt_info[$i];
    $cofbt[$i] = $filt_info[$filt_leng + $i + 1];
}
print @cofat;
print @cofbt;


# Read in second argument as name for output stat file
$STAT_NAME = $ARGV[1];
chomp($STAT_NAME); #get rid of pesky \n

# Read in thrid and fourth arguments and assign as bit-width values
my $argck2 = $ARGV[2];
chomp($argck2);
if ($argck2 = -i)
{$bw_in = $ARGV[3];}
#$bw_in = $ARGV[3];
#chomp($bw_in);
elsif ($argck2 = -o)
{$bw_out = $ARGV[3];}

#my $argck4 = $ARGV[4];
#chomp($argck4);
#if ($argck4 = -o)
#{$bw_out = $ARGV[5];}
#elsif ($argck4 = -i)
#{$bw_in = $ARGV[5];}
$bw_out = $ARGV[5];
chomp($bw_out);

# Assign insertable variables
```

```perl
$mfl = $filt_leng;

$i = $filt_leng; $lvs = 1;
while($i > 2){
    $i = $i/2;
    $lvs++;}

$bwi = $bw_in;
$bwc = $bw_coeff;
$bwmp = abs($bw_in - $bw_coeff);
#$bwo = $bw_out;
$bwo = $bwi + $bwc + $lvs;

$bsi = '"';
for($i=0;$i<$bwi;$i++){
    $bsi = $bsi . '0';
} $bsi = $bsi . '"';

$bsc = '"';
for($i=0;$i<$bwc;$i++){
    $bsc = $bsc . '0';
} $bsc = $bsc . '"';

$bsic = '"';
for($i=0;$i<($bwi+$bwc);$i++){
    $bsic = $bsic . '0';
} $bsic = $bsic . '"';

$bso = '"';
for($i=0;$i<$bwo;$i++){
    $bso = $bso . '0';
} $bso = $bso . '"';

$cii = '"';
for($i=0;$i<($bwo-1);$i++){
    $cii = $cii . '0';
} $cii = $cii . '"';

if($bwmp == 0)
{$psm = "pad_string_mult   :   std_logic := '0';";}
elsif($bwmp == 1)
{$psm = "pad_string_mult   :   std_logic := '0';";}
else
{$psm = "pad_string_mult   :   std_logic_vector(";
 $psm = $psm . ($bwmp-1);
 $psm = $psm . " downto 0) := ";
 $psm = $psm . '"';
 for($i=0;$i<$bwmp;$i++){
     $psm = $psm . '0';
 } $psm = $psm . '"';
}

if($lvs == 1)
{$psa = '0';}
else
{$psa = '"';
 for($i=0;$i<$lvs;$i++){
     $psa = $psa . '0';
 } $psa = $psa . '"';
}
```

213

```perl
if($bwi >= $bwc)
{$ext = (($mfl+1)*($bwi+1));}
else
{$ext = (($mfl+1)*($bwc+1));}


# Give values to template variables
$pipe_in_l = @pipe_in;
for($i=0;$i<$pipe_in_l;$i++){
    $pipe_in[$i] =~ s/--mfl--/$mfl/g;
    $pipe_in[$i] =~ s/--lvs--/$lvs/g;
    $pipe_in[$i] =~ s/--bwi--/$bwi/g;
    $pipe_in[$i] =~ s/--bwc--/$bwc/g;
    $pipe_in[$i] =~ s/--bwmp--/$bwmp/g;
    $pipe_in[$i] =~ s/--bwo--/$bwo/g;
    $pipe_in[$i] =~ s/--bsi--/$bsi/g;
    $pipe_in[$i] =~ s/--bsin--/$bwi-1/ge;
    $pipe_in[$i] =~ s/--bsc--/$bsc/g;
    $pipe_in[$i] =~ s/--bscn--/$bwc-1/ge;
    $pipe_in[$i] =~ s/--bsic--/$bsic/g;
    $pipe_in[$i] =~ s/--bsicn--/$bwi+$bwc-1/ge;
    $pipe_in[$i] =~ s/--bso--/$bso/g;
    $pipe_in[$i] =~ s/--bson--/$bwo-1/ge;
    $pipe_in[$i] =~ s/--cii--/$cii/g;
    $pipe_in[$i] =~ s/--ciin--/$bwo-1/ge;
    $pipe_in[$i] =~ s/--psm--/$psm/g;
    $pipe_in[$i] =~ s/--psa--/$psa/g;
    $pipe_in[$i] =~ s/--psan--/$lvs-1/ge;
    $pipe_in[$i] =~ s/--ext--/$ext/g;

    $pipe_in[$i] =~ s/--cofat--/@cofat/g;
    $pipe_in[$i] =~ s/--cofbt--/@cofbt/g;
}

$sequ_in_l = @sequ_in;
for($i=0;$i<$sequ_in_l;$i++){
    $sequ_in[$i] =~ s/--mfl--/$mfl/g;
    $sequ_in[$i] =~ s/--lvs--/$lvs/g;
    $sequ_in[$i] =~ s/--bwi--/$bwi/g;
    $sequ_in[$i] =~ s/--bwc--/$bwc/g;
    $sequ_in[$i] =~ s/--bwmp--/$bwmp/g;
    $sequ_in[$i] =~ s/--bwo--/$bwo/g;
    $sequ_in[$i] =~ s/--bsi--/$bsi/g;
    $sequ_in[$i] =~ s/--bsin--/$bwi-1/ge;
    $sequ_in[$i] =~ s/--bsc--/$bsc/g;
    $sequ_in[$i] =~ s/--bscn--/$bwc-1/ge;
    $sequ_in[$i] =~ s/--bsic--/$bsic/g;
    $sequ_in[$i] =~ s/--bsicn--/$bwi+$bwc-1/ge;
    $sequ_in[$i] =~ s/--bso--/$bso/g;
    $sequ_in[$i] =~ s/--bson--/$bwo-1/ge;
    $sequ_in[$i] =~ s/--cii--/$cii/g;
    $sequ_in[$i] =~ s/--ciin--/$bwo-1/ge;
    $sequ_in[$i] =~ s/--psm--/$psm/g;
    $sequ_in[$i] =~ s/--psa--/$psa/g;
    $sequ_in[$i] =~ s/--psan--/$lvs-1/ge;
    $sequ_in[$i] =~ s/--ext--/$ext/g;

    $sequ_in[$i] =~ s/--cofat--/@cofat/g;
    $sequ_in[$i] =~ s/--cofbt--/@cofbt/g;
}
```

```perl
# Create Stats Text
my $full_fl = $filt_leng*2;
@stats = "Special Sampling Scheme I/Q Downconverter\nFilter Length =
$full_fl\nInput Bit-Width = $bw_in\nCoefficient Bit-Width = $bw_coeff\nOutput
Bit-Width = $bw_out\n";


# Print Out all output files
$STAT_OUT = $STAT_NAME . ".stat";
open($OUT_STAT, "> $STAT_OUT") || die("cannot create $STAT_OUT: $!");
print $OUT_STAT @stats;
close($OUT_STAT);

$PIPE_OUT = 'sss_pipeline.vhd';
open($OUT_PIPE, "> $PIPE_OUT") || die("cannot create $PIPE_OUT: $!");
print $OUT_PIPE @pipe_in;
close($OUT_PIPE);

$SEQU_OUT = 'sss_sequential.vhd';
open($OUT_SEQU, "> $SEQU_OUT") || die("cannot create $SEQU_OUT: $!");
print $OUT_SEQU @sequ_in;
close($OUT_SEQU);

#$ABST_OUT = 'sss_abstract.vhd';
#open($OUT_ABST, "> $ABST_OUT") || die("cannot create $ABST_OUT: $!");
#print $OUT_ABST @abst_in;
#close($OUT_ABST);
```

Appendix R: tb_gen.prl (Perl)

```perl
#!/usr/sbin/perl -w

#use 5.004;
#use strict;          # enforce declarations and quoting
#use CGI qw(:standard); # import shortcuts
#use Fcntl qw(:flock);  # imports LOCK_EX, LOCK_SH, LOCK_NB

my(
    $MATLAB_IN, # name of filter coefficient .m input file
    $PIPE_IN, # name of pipelined vhdl input file
    $PIPE_OUT, # name of pipelined vhdl output file
    $SEQU_IN, # name of sequential vhdl input file
    $SEQU_OUT, # name of sequential vhdl output file
    $ABST_IN, # name of abstract vhdl input file
    $ABST_OUT, # name of abstract vhdl output file

    @input_info, # array to hold info of input Matlab txt file
    @pipe_in, #array holding pipeline template
    $pipe_in_l, #length of the array
    @sequ_in, #array holding sequential template
    $sequ_in_l, #length of the array
    @abst_in, #array holding abstract template
    $abst_in_l, #length of the array
    $bw_in, # bit width of the input
    $bw_out, # bit width of the output

    # variables for input to template files
    $bwi, $bwo, $ext,

    $i, # a counter variable for for loops
    $first,

);

# usage: gen_tb.prl [matlab_file] [-i] [bit_width_in] [-o] [bit_with_out] [-s]
[-p] [-a]
# $TITLE = "Replace template VHDL files with parameterized versions";
# This program will replace generic varibles in the templates with user defined
values for filter length and bit_widths.
# operation: "gen_tb.prl

# print ("\nEnter the name of the Original Fault File : ");


# Put input templates into info arrays
# open the input files for read (preserving old contents)
$PIPE_IN = "tb_pipeline_template.vhd";
chomp($PIPE_IN); #get rid of pesky \n
open(IN_PIPE, "< $PIPE_IN") || die("cannot open $PIPE_IN: $!");
@pipe_in=<IN_PIPE>;
close(IN_PIPE);

$SEQU_IN = "tb_sequential_template.vhd";
chomp($SEQU_IN); #get rid of pesky \n
open(IN_SEQU, "< $SEQU_IN") || die("cannot open $SEQU_IN: $!");
@sequ_in=<IN_SEQU>;
close(IN_SEQU);
```

```perl
$ABST_IN = "tb_abstract_template.vhd";
chomp($ABST_IN); #get rid of pesky \n
open(IN_ABST, "< $ABST_IN") || die("cannot open $ABST_IN: $!");
@abst_in=<IN_ABST>;
close(IN_ABST);


# Read in first agument as input file and put into info array
$MATLAB_IN = $ARGV[0];
chomp($MATLAB_IN); #get rid of pesky \n
open(IN_MATLAB, "< $MATLAB_IN") || die("cannot open $MATLAB_IN: $!");
@input_info=<IN_MATLAB>;
close(IN_MATLAB);

# Read in thrid and fourth arguments and assign as bit-width values
my $argck1 = $ARGV[1];
chomp($argck1);
if ($argck1 = -i)
{$bw_in = $ARGV[2];}
#$bw_in = $ARGV[2];
#chomp($bw_in);
elsif ($argck1 = -o)
{$bw_out = $ARGV[2];}

#my $argck3 = $ARGV[3];
#chomp($argck3);
#if ($argck3 = -o)
#{$bw_out = $ARGV[4];}
#elsif ($argck3 = -i)
#{$bw_in = $ARGV[4];}
$bw_out = $ARGV[4];
chomp($bw_out);

# Assign insertable variables
$bwi = $bw_in;
$bwo = $bw_out;
#$bwo = $bwi + $bwc + $lvs;

#if($bwi >= $bwc)
#{$ext = (($mfl+1)*($bwi+1));}
#else
#{$ext = (($mfl+1)*($bwc+1));}


# Give values to template variables
$pipe_in_l = @pipe_in;
for($i=0;$i<pipe_in_l;$i++){
    $pipe_in[$i] =~ s/--bwi--/$bwi/g;
    $pipe_in[$i] =~ s/--bwo--/$bwo/g;
    $pipe_in[$i] =~ s/--bsin--/$bwi-1/ge;
    $pipe_in[$i] =~ s/--bson--/$bwo-1/ge;

    $pipe_in[$i] =~ s/--input--/@input_info/g;
}

$abst_in_l = @abst_in;
for($i=0;$i<$abst_in_l;$i++){
    $abst_in[$i] =~ s/--bwi--/$bwi/g;
    $abst_in[$i] =~ s/--bwo--/$bwo/g;
    $abst_in[$i] =~ s/--bsin--/$bwi-1/ge;
    $abst_in[$i] =~ s/--bson--/$bwo-1/ge;
```

217

```perl
    $abst_in[$i] =~ s/--input--/@input_info/g;
}

$sequ_in_l = @sequ_in;
for($i=0;$i<$sequ_in_l;$i++){
    $sequ_in[$i] =~ s/--bwi--/$bwi/g;
    $sequ_in[$i] =~ s/--bwo--/$bwo/g;
    $sequ_in[$i] =~ s/--bsin--/$bwi-1/ge;
    $sequ_in[$i] =~ s/--bson--/$bwo-1/ge;

    $sequ_in[$i] =~ s/--input--/@input_info/g;
}


$PIPE_OUT = 'tb_pipeline.vhd';
open($OUT_PIPE, "> $PIPE_OUT") || die("cannot create $PIPE_OUT: $!");
print $OUT_PIPE @pipe_in;
close($OUT_PIPE);

$SEQU_OUT = 'tb_sequential.vhd';
open($OUT_SEQU, "> $SEQU_OUT") || die("cannot create $SEQU_OUT: $!");
print $OUT_SEQU @sequ_in;
close($OUT_SEQU);

$ABST_OUT = 'tb_abstract.vhd';
open($OUT_ABST, "> $ABST_OUT") || die("cannot create $ABST_OUT: $!");
print $OUT_ABST @abst_in;
close($OUT_ABST);
```

Vita

       Lieutenant Jesse Somann graduated from Hartford Union High School in Hartford, WI in 2000. He attended Rose-Hulman Institute of Technology under an AFROTC scholarship as and Electrical Engineering student. He graduated with a BS in Electrical Engineering and a minor in Geography, with honors and was commissioned in May, 2004.

       Lt. Somann's first assignment is here at the Air Force Institute of Technology to complete a Master's Degree in Electrical Engineering. He will graduate in March of 2006 and will take a follow on assignment here at Wright-Patterson AFB as a developmental engineer with the Air Force Research Labs

REPORT DOCUMENTATION PAGE

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) 23-03-2006	2. REPORT TYPE **Master's Thesis**	3. DATES COVERED (From – To) Aug 2004-Mar 2006

4. TITLE AND SUBTITLE CHARACTERIZATION AND DESIGN OF HIGH-LEVEL VHDL I/Q FREQUENCY DOWNCONVERTER VIA SPECIAL SAMPLING SCHEME	5a. CONTRACT NUMBER
	5b. GRANT NUMBER 2005-088
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Somann, Jesse P., Second Lieutenant, USAF	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way Bldg 640 WPAFB OH 45433-7765	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GE/ENG/06-53

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/SNDI ATTN: Dr. Gregory Creech 2241 Avionics Cir. WPAFB, OH 45433 Comm:937-255-4831 ext:3486 DSN: 785-4831 ext:3486 Email: Gregory.Creech@wpafb.af.mil	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This study explores the characterization and implementation of a Special Sampling Scheme (SSS) for In-Phase and Quad-Phase (I/Q) downconversion utilizing top-level, portable design strategies. The SSS is an under-developed signal sampling methodology that can be used with military and industry receiver systems, specifically, United States Air Force (USAF) video receiver systems. The SSS processes a digital input signal-stream sampled at a specified sampling frequency, and downconverts it into In-Phase (I) and Quad-Phase (Q) output signal-streams. Using the theory and application of the SSS, there are three main objectives that will be accomplished: characterization of the effects of input, output, and filter coefficient parameters on the I/Q imbalances using the SSS; development and verification of abstract, top-level VHDL code of the I/Q SSS for hardware implementation; and finally, development, verification, and analysis of variation between synthesizable pipelined and sequential VHDL implementations of the SSS for Field Programmable Gate Arrays (FPGA) and Application Specific Integrated Circuits (ASIC).

15. SUBJECT TERMS

I/Q Downconversion, VHDL, Special Sampling Scheme, Abstract, Characterization

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON **Yong C. Kim, PhD (ENG)**
REPORT **U**	ABSTRACT **U**	c. THIS PAGE **U**	**UU**	221	19b. TELEPHONE NUMBER (Include area code) (937) 255-3636 ext 4620; email: yong.kim@afit.edu

Standard Form 298

(Rev. 8/98)